СОДЕРЖАНИЕ

ВВЕДЕНИЕ....7
Компьютерные игры....7
Об этой книге....7
Что вы должны знать....7
Как организована эта книга....7
ВИДЕОИГРЫ. ПЕРВЫЕ ШАГИ....9
Кто пишет видеоигры?....9
Откуда берутся идеи?....9
Фазы создания видеоигр....9
Что вы узнаете из этой книги?....10
ИТОГ....10
ОСНОВЫ ЯЗЫКА АССЕМБЛЕРА....11
Зачем нам нужен ассемблер при написании игр?....11
Обзор семейства процессоров 80x86....11
Регистры процессора....12
Флаговый регистр....12
Общий вид ассемблерной функции....12
Передача параметров....14
Директива USES....15
Внимание!....15
Передача указателей....16
Локальные переменные....16
Директива LOCAL....17
Создание внешних ссылок....18
Возвращение параметров в вызывающую функции....18
Установка видеорежимов....19
Сверхскоростная очистка экрана....20
Использование встроенного (in-line) ассемблера....21
ИТОГ....22
ОСНОВЫ РАБОТЫ С УСТРОЙСТВАМИ ВВОДА....24
Взаимодействие с пользователем в видеоиграх....24
Джойстик....24
Как работает джойстик....24
Кнопки джойстика....25
Чтение позиции джойстика....26
Калибровка джойстика....27
Клавиатура....31
Скан-коды....31
Статус клавиш....32
Получение скан-кодов с клавиатуры....33
Получение ASCII-кодов с клавиатуры....33
Теперь все вместе: Демонстрационная программа работы с клавиатурой....33
Мышь....36
ИТОГ....40
МЕХАНИЗМЫ ДВУХМЕРНОЙ ГРАФИКИ....41
Использование матриц. Двухмерная проекция....41
Точки, линии и области....41
Точки....41
Линии....42
Многоугольники....42
Объекты....43
Позиционирование объекта....44
Трансляция объектов....45
Масштабирование объектов....46
Вращение объектов....46

Отсечения 51
Матрицы 51
Произведение операций над матрицами 52
Единичная матрица 53
Использование матриц в играх 53
Главная матрица перемещений 53
Главная матрица масштабирования 53
Главная матрица поворотов 54
Общая матрица масштабирования, поворотов и перемещений 54
Компонент нормализации вершины 54
Программа Астероиды с использованием матриц 54
К вашему сведению 58
Основы контроля столкновений 59
ИТОГ 59
СЕКРЕТЫ VGA-КАРТ 60
Обзор VGA-карт 60
256-цветный режим 60
Конфигурация видеопамяти 61
Таблицы цветов 61
Переопределение цветовой палитры 62
Цветовая ротация 64
Рисование точки 65
Рисование линий 66
Графический формат PCX 69
Побитовое копирование изображения (бит-блиттинг) 74
Спрайты 74
Отображение текста 81
Дублирующее буферизирование 83
Вертикальный обратный ход луча 84
Тайминг 85
Игра Tombstone 85
Итог 93
ТРЕТЬЕ ИЗМЕРЕНИЕ 94
Что такое трехмерное пространство 94
Точки, линии, многоугольники и объекты в трехмерном пространстве 95
Перемещения, масштабирование и повороты в трехмерном пространстве 98
Перемещение трехмерного объекта 98
Масштабирование трехмерного объекта 98
Вращение трехмерного объекта 98
Вращение, параллельное оси X 98
Вращение, параллельное оси Z 99
Последнее слово о трехмерных трансформациях 99
Проекции 99
Копирование на экран 101
Масштабирование экрана 102
Математические основы параллельных проекций 103
Математические основы верспективной проекции 103
Объем просмотра 103
Геометрическое моделирование 104
Удаление невидимых поверхностей 104
Алгоритм художника 105
Алгоритм Z - буфера 108
Уравнение плоскости 109
Использование уравнения плоскости для вершин многоугольника 109
Соотношение пространства образов и пространства объектов 109
Трассировка луней 109

Отсечение лучей.....110
Математические основы отсечения лучей.....115
Рисование лучей.....115
Определение значения наклона.....115
Вычисление линий для генерации.....116
Вычисление точки первого пересечения.....116
Вычисление оставшихся пересечений.....117
Вычисление расстояния.....119
Вычисление масштаба.....120
Уменьшение проекционных искажений.....120
Отрисовка фрагментов стен.....121
Реализация отсектеля лучей.....121
Оптимизация отсекателя лучей.....126
Отрисовка дверей и прозрачных областей.....126
Освещение, тени и палитра.....127
Рассеянное освещение.....127
Локальное освещение.....128
Закон Ламберта.....128
Закон обратных квадратов.....129
Создание модели освещения.....129
ИТОГ.....131
УЛУЧШЕННАЯ БИТОВАЯ ГРАФИКА И СПЕЦИАЛЬНЫЕ ЭФФЕКТЫ.....132
Ускорение процесса двоичного кодового преооразовзния (бит-блиттинга).....132
Применение логических операций.....133
Операция.....134
Кодирование прозрачности.....136
Битовое отсечение.....139
Контроль столкновения спрайтов.....142
Дублирующая буферизация.....144
Использование сигнала вертикальной синхронизации.....146
Мультипликация с помощью регистров цвета.....148
Освещение ваших игр.....150
Связь мультипликации с контекстом.....151
«Animotion».....152
Специальные эффекты.....157
Фоновая мультипликация.....157
Экранные эффекты.....159
Гаснущее изображение.....159
Исчезновение Изображения.....159
Оплывание изображения.....160
Демонстрация смены экрана.....160
Пробовали ли вы текстурировать?.....163
Масштабирование растровых изображений.....163
Повороты объектов.....167
Итог.....171
ВЫСОКОСКОРОСТНЫЕ ТРЕХМЕРНЫЕ СПРАЙТЫ.....172
Механика трехмерных спрайтов.....172
Аксонометрические преобразования.....174
Правильный расчет масштаба.....175
Видимый объем.....175
Новая версия масштабирования.....176
Отсечение спрайтов в трехмерном пространстве.....182
Построение траекторий.....183
Удачный угол зрения на спрайты.....185
Трехмерное звездное небо.....188
Реализм.....189

Создание звездного неба с использованием целых чисел....189
Оцифровка объектов и моделирование....196
Создание съемочного мини-павильона....197
Фон....197
Платформа....197
Освещение....198
Создание студии....198
Video for Windows....198
Цветовая палитра....199
Эксперименты с макетами....200
Коротко! (То есть итог)....201
ЗВУКОВЫЕ ЭФФЕКТЫ И МУЗЫКА....202
Устройство и архитектура звуковой карты Sound Blaster....205
Оцифрованный звук....207
Воспроизведение оцифрованного звука....208
Описание....209
Использование функций драйвера для проигрывания VOC-файлов....210
Частотный синтезатор....218
Музыка И MIDI....218
Использование звука в играх....219
ИТОГ....221
РАЗРАБОТКА МУЗЫКИ ДЛЯ КОМПЬЮТЕРНЫХ ИГР....222
Способы воспроизведения звука....224
Пакеты программ DIGPAK и MIDPAK....227
Что такое «расширенный MIDI-формат»?....227
В каком формате должны быть представлены данные MIDI?....227
Вопросы и ответы....228
API пакета программ DIGPAK....228
Функция № 1: DigPlay....229
Функция № 2: SoundStatus....229
Функция № 3: MassageAudio....230
Функция № 4: DigPlay2....230
Функция № 5: AudioCapabilities....230
Функция № 8: StopSound....230
Функция № 12: SetTimerDivisorRate....231
Функция № 14: PostAudioPending....231
Функция № 15: AudioPendingStatus....231
Функция № 16: SetStereoPan....232
Функция № 17: SetPlayMode....232
Функция № 18: Адрес флага ожидания и адрес «семафора» DIGPAK....232
Функция № 19: Установка режима записи....233
Функция № 21: Установка режима обратной записи DMA....233
Функция № 22: Значение счетчика прямого доступа в память....234
Функция № 23: Проверка буфера прямого доступа в память....234
Функция № 24: Установка громкости при импульсно-кодовой модуляции....234
Функция № 25: SetDPMIMode....234
API пакета программ MIDPAK....237
Функция № 1: UnloadMidPak....238
Функция № 2: DigPakAvailable....238
Функция № 3 PlaySequence....238
Функция № 4: SegueSequence....238
Функция № 5: RegisterXmidi....238
Функция № 6: MidiStop....238
Функция № 8: ReportTriggerCount....238
Функция № 9: ResetTriggerCount....239
Функция № 12: ResumePlaying....239

Функция № 13: SequenceStatus.....239
Функция № 14: RegisterXmidiFile.....239
Функция № 15: RelativeVolume.....239
Функция № 16: SetRelativeVolume.....239
Функция № 17: BootstrapMidPak.....239
Функция № 18: PollMidPak.....239
Функция № 19: MidpakClock.....240
Функция № 20: TriggerCountAddress.....240
Функция № 21: EventIDAddress.....240
Функция № 23: ReportSequenceNumber.....240
Завтрак с Толстяком - музыка в программах.....241
Почему?.....241
Когда?.....242
Как?.....242
Создание пользовательского интерфейса с системой распознавания голоса.....243
Демонстрационная программа Test.c.....244
АЛГОРИТМЫ, СТРУКТУРЫ ДАННЫХ И МЕТОДОЛОГИЯ ВИДЕОИГР.....247
Структуры данных, используемые для представления игрового пространства.....248
Когда объекты сталкиваются.....253
Игровые объекты.....254
Структуры данных в компьютерных играх.....254
Планирование игровых объектов.....256
Состояния бытия.....257
Интерфейс пользователя.....260
Демонстрационный режим.....261
Сохранение игры.....262
Моделирование реального мира.....263
ИТОГ.....268
Мнимое время, прерывания и многозадачность.....269
Восприятие игры.....269
Что такое многозадачность?.....269
Реализация многозадачности при помощи прерываний.....271
Внимание!.....273
Написание программы обработки прерываний на языке Си.....273
Установка обработчика прерывания.....273
Игровой цикл.....275
Автономные функции.....277
Функции ответа.....278
Программируем системные часы.....281
Изменение частоты таймера.....282
Значение.....282
Десятичное.....282
Частота.....282
Собираем все вместе.....284
Пример обработчика прерывания № 1 - Там полно звезд.....285
Пример обработчика прерывания № 2 - Ловим нажатия клавиш!.....288
Заключение.....291
ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ.....292
Как мыслят видеоигры: обзор.....292
Алгоритмы Преследования и Уклонения.....293
Преследование.....293
Уклонение.....295
Шаблонные мысли.....295
Случайные передвижения.....299
Конечные автоматы.....301
Конечный автомат, управляемый окружающей средой.....303

Управление приоритетным состоянием.....307
Вероятностные автоматы.....307
Память и обучение.....311
Алгоритмы Поиска. Выслеживание игрока.....311
Теория игр.....313
Итог.....314
Связь.....315
Средства связи видеоигр.....315
Последовательный интерфейс ПК.....315
Универсальный асинхронный приемопередатчик.....316
Установки и статус UART.....316
Аппаратное обеспечение UART.....320
Операции с ROM BIOS.....321
Соединение через нуль-модем.....321
Построение коммуникационной библиотеки.....322
Инициализация последовательного порта.....322
Установка прерывания.....324
Чтение символа из буфера.....325
Запись в последовательный порт.....326
Коммуникационная программа: NLINK.....326
Стратегия игровых коммуникаций.....330
Синхронизация вектора состояния.....332
Синхронизация состояния ввода/вывода.....334
Временная синхронизация.....335
Модем.....336
Net Tank: танковый имитатор для двух игроков.....337
Анализ игры Net-Tank.....338
Раздел 1: Инициализация.....338
Раздел 2: Игровой цикл.....338
Раздел 3: Удаление объектов.....338
Раздел 4: Получение входных данных и передача состояния дистанционно управляемой системе.....338
Раздел 5: Перемещение объектов.....339
Раздел 6: Распознавание столкновений.....339
Раздел 7: Рисование объектов.....339
Раздел 8: Дублирующий буфер.....340
Раздел 9: Всякая всячина.....340
Раздел 10: Опять и снова к опять.....340
Итог.....340
ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА.....341
Определение понятия инструментального средства.....341
Редакторы изображений.....342
Пакеты анимации.....342
Производство кинофильма.....342
Звуковые редакторы.....343
Редактор поля WarEdit.....343
Использование WarEdit.....345
Функциональное описание программы WarEdit.....345
Улучшения WarEdit.....348
Итог.....349
ГРАФИЧЕСКОЕ ОФОРМЛЕНИЕ ИГРЫ.....350
Удобные детали и инструменты.....350
Первый этап: планирование.....351
Разрешающая способность и количество цветов.....351
Для кого вы пишете игру?.....351
Содержание и общее настроение игры.....351
Что детализировать, а что нет?.....351

Несколько слов по поводу цвета... 351
Разработка 256-цветной палитры... 352
Ну дайте мне место (разработка стен, пола и «кирпичиков» потолка!... 356
Детали, детали и детали... 356
Придание «кирпичикам» стен глубины трехмерного пространства... 359
Масштабирование изображения... 359
Создание бесшовных мозаичных изображений... 360
Особые ситуации... 361
Удаление неровностей... 365
Как избежать цветовых проблем во время игры... 365
Если б только у меня было оружие и еда! (разработка объекта)... 365
Соотношение размеров объектов и стен... 366
Помните о перспективе... 366
Подайте сюда врагов! (разработка персонажей)... 366
Существует ли легкий путь?... 366
Пропорции... 366
Основополагающие принципы для оживления персонажа... 367
Технические приемы анимации... 367
Чудовища и прочие кровожадные персонажи... 367
Последние дорисовки... 368
Начальная и конечная заставки... 368
Оформление оценочного экрана... 368
Экраны для отражения статуса и изменения конфигурации игры... 369
Анимация... 369
Кадры для привлечения внимания... 369
ИТОГ... 369
ТЕХНИКА СОЗДАНИЙ ПАРАЛЛАКСА... 370
Еще кое-что о режиме 13h... 370
Примечание по поводу демонстрационной программы... 370
Внимание... 370
Первый шаг... 371
Несколько смещающихся слоев... 377
«Прозрачные» пиксели... 378
Оптимизированные версии OpaqueBlt() и TransparentBlt()... 383
Смещение мозаичных изображений... 384
Устранение эффекта сдвига кадра... 391
PCX-графика... 391
Примечания по выполнении... 391
Примечание... 392
ИТОГ... 392
ТЕХНИКА ОПТИМИЗАЦИИ... 393
Введение... 393
Передача параметров... 393
Глобальные переменные... 394
Указатели и использование псевдоимен... 394
Использование регистров... 395
Оптимизация компилятора... 395
Развертка циклов... 396
Бинарное умножение... 397
Таблицы поиска... 398
Математика с фиксированной запятой... 400
Присваивание... 401
Сложение и вычитание... 401
Умножение... 402
Деление... 402
Точность... 402

Максимальное цифровое представшие.....402
Встроенный ассемблер.....404
Оптимизация рисования пикселей.....404
Оптимизация изображаемой линии.....406
ИТОГ.....407
ИГРА WARLOCK (КОЛДУН).....408
Сюжет игры.....410
Компоненты игры.....410
Примечание по звуку.....410
Загрузка и компиляция Warlock.....411
Режим демонстрации.....411
Движение по игровому пространству.....412
Новый отсекатель лучей.....412
Секция 1.....416
Секция 2.....416
Секция 3.....416
Секция 4.....416
Секция 5.....416
Секция 6.....416
Секция 7.....416
Изображение текстуры.....416
Оттенение.....419
Использование ассемблера.....419
Цикл игры.....420
Игровое пространство.....421
Режим демонстрации.....423
Размещение объектов в пространстве.....423
Быстрее, быстрее, еще быстрее.....423
Несколько слов напоследок.....424

ВВЕДЕНИЕ

Как претворить в жизнь идею компьютерной игры? Приходилось ли вам, играя в свою любимую игру, мечтать о том, как можно было бы ее улучшить? Задумывались ли вы в процессе игры о том, как она устроена? Эта книга откроет вам путь к созданию собственных игр.

Компьютерные игры

Если вы не играли в свою любимую аркадную игру и не убивали кучу монстров уже несколько месяцев — вы явно заработались! Вернитесь к играм и увидите, как вы помолодеете и как разгладятся морщины на вашем утомленном лице. Дни Hunt the Wampus и Lunar Lander миновали, и теперь игры стали намного ярче, красочней и хитрее.

Игра сегодняшнего дня содержит целые мили туннелей, дорог и странных созданий. Они сопровождаются потрясающей графикой и звуком. DOOM - хороший пример такой игры. Она включает запутанные переходы и тайники с сокровищами, чудовищ, на которых вы можете охотиться и которых вы должны убить прежде, чем они убьют вас. Если вы хоть раз играли в DOOM, то не перестанете тосковать по кислотным лужам, ружьям, чудовищам, скрытым проходам, лифтам, взрывам... этот список можно продолжать до бесконечности. Все это было создано благодаря комбинации графики и звукового сопровождения.

Графика DOOM дает вам полное ощущение пространства и перспективы — как если бы вы действительно находились в центре событий. Для этого DOOM использует приемы работы с трехмерной графикой. Разработчики игр для ПК используют трехмерную графику для увеличения реалистичности игры любого типа — посмотрите на 7th Guest, MYST, X-Wing, Outpost, Indy Car Racing. Часто трехмерная графика применяется и в имитаторах спортивных игр, таких как скачки или бокс.

После добавления к такой графике сюжета и высококачественного звука, разработку игры можно считать законченной. Вы почувствуете себя реальным участником действия — стрельбы, гонок и поиска сокровищ, прилагающим все усилия, чтобы остаться в живых.

Об этой книге

Большинство книг по компьютерным играм объясняют, как в них играть. В отличие от них, эта книга учит, как создавать собственные игры, используя опыт, накопленный профессионалами. Профессиональные разработчики игр для ПК научат вас, как создать игру, использующую высококачественную графику и звук. Если вы только начинаете карьеру разработчика компьютерных игр, эта книга окажется для вас незаменимым подспорьем. Она научит вас, как осуществлять прокрутку графики, оформлять таблицы лучших результатов, программировать трехмерную графику, создавать текстуры и многому, многому другому. Если вы уже имеете опыт в разработке игр, эта книга поможет вам освоить массу новых технических приемов.

Что вы должны знать

Для понимания большинства материала, представленного в этой книге, вы должны хорошо владеть языком программирования Си. Большинство книг по созданию компьютерных игр ориентируются на ассемблер. В этой книге ассемблеру уделено совсем немного внимания, а основной акцент сделан на Си. Однако, она содержит несколько ассемблерных примеров, так что знакомство с этим языком вам не помешает.

Как организована эта книга

Эта книга посвящена технике разработки игр для ПК. Она не является учебником, в котором материал представлен последовательно — от простого к сложному, однако в построении книги заметна определенная система.

В первой главе, «Видеоигры, первые шаги...», вы увидите, кто пишет игры, откуда берутся идеи и изучите основные этапы разработки игровой программы,

Вторая глава, «Основы языка ассемблера», даст вам небольшой урок по ассемблеру. Вы изучите терминологию и методы, использованные при создании игр, представленных в этой книге.

В третьей главе, «Основы устройств ввода», будет дан обзор методов построения интерфейса между играми и такими устройствами, как джойстик, мышь и клавиатура.

Четвертая глава, «Механизмы двухмерной графики», покажет вам основные приемы, используемые при разработке двухмерных игр типа Commander Keen, Вы изучите технику вращения, масштабирования, перемещения и отсечения объектов в двухмерных играх.

Глава пятая, «Секреты VGA-карт», посвящена работе с графикой VGA. Вы изучите 256-

цветный режим, таблицу цветов, копирование битовых изображений и т. д.

Шестая глава, «Третье измерение», укажет вам путь в трехмерный мир- Вы узнаете, как создавать DOOM-подобные миры, используя технику работы с трехмерной графикой.

Седьмая глава, «Усовершенствованная битовая графика и специальные эффекты», позволит вам получить представление об ускоренном выводе изображений, работе со светотенью и другими специальными эффектами. Будьте внимательны — отсюда начинаются боевики!

Глава восьмая, «Высокоскоростные трехмерные спрайты», откроет вам дорогу в изумительный мир высокоскоростных трехмерных спрайтов. Эта глава покажет, как создать реалистичную модель космического корабля и вставить ее в вашу программу.

В девятой главе, «Звуковые эффекты и музыка», вы познакомитесь с основами применения звука в ваших играх. Вы сможете вызвать у игрока волнение и ужас, используя Sound Blaster. В игре музыка заставит ваше сердце биться в такт происходящему.

В десятой главе, «Разработка музыки для компьютерных игр» вы изучите приемы работы с программами DIGPAK и MIDPAK, которые позволяют создавать впечатляющую музыку и звуковые эффекты. Благодаря этим программам вы сможете добавлять оркестровую музыку и необычайные звуковые эффекты в ваши творения, что придаст им профессиональный вид. Эти программы использовались при создании таких популярных игр, как 7th Guest, Terminator 2029, Mechwarrior II.

Глава одиннадцатая, «Алгоритмы, структуры данных и методология видеоигр», знакомит вас с внутренней структурой профессиональных игр. Вы научитесь моделировать реальный мир в своих виртуальных вселенных.

Двенадцатая глава, «Мнимое время, прерывания и многозадачность», научит вас добавлять новое измерение — время. Вы изучите приемы параллельной обработки различных игровых задач, создания главного игрового цикла и как использовать прерывания при работе с устройствами ввода/вывода.

Глава тринадцатая, «Искусственный интеллект», поможет вам сделать своих персонажей более интеллектуальными. Вы же не хотите, чтобы они напрасно теряли время. Научите их думать и реагировать на ваши действия так, чтобы игру невозможно было отличить от реальности.

Четырнадцатая глава, «Связь», научит вас добавлять дополнительные возможности. Вы сможете звякнуть по модему приятелям и отправиться вместе с ними по неисследованным дебрям виртуальных миров. Игры, рассчитанные на нескольких участников, становятся все более популярными.

Глава пятнадцатая, «Инструментальные средства», даст вам полный набор инструментов для построения ваших игр. С помощью этих инструментов можно создать текстуры, персонажей, звуковые файлы, планы игрового поля, декорации.

В шестнадцатой главе, «Графическое оформление игр», вы получите представление о точке зрения художника. Используя подходы, описанные в этой главе, вы сможете придать игре больший профессионализм и художественную ценность.

Семнадцатая глава, «Техника создания параллакса», учит, как заставить удаленные объекты двигаться медленнее, чем приближенные. Эта техника позволит оживить перспективу в ваших играх.

В восемнадцатой главе, «Техника оптимизации», демонстрируются технические приемы повышения производительности. Ваша игра будет работать быстрее и движения персонажей станут более плавными, если вы последуете приведенным в этой главе советам.

Девятнадцатая глава, "Игра Warlock (Колдун)", описывает игру, созданную автором этой книги.

ВИДЕОИГРЫ. ПЕРВЫЕ ШАГИ...

С чего начать? Хочется так много сказать, что невольно придется посвятить этому несколько страниц. То путешествие, которое мы собираемся предпринять в мир разработки видеоигр, можно смело назвать захватывающим приключением. Создание видеоигр можно сравнить с написанием стихов или рисованием картины. Для этого нужно вдохновение, ведь создатель хочет поделиться с окружающим миром частичкой своего воображения. Один великий скульптор сказал однажды; «Статуя была здесь всегда, Я просто освободил ее из камня». Это высказывание вполне применимо и к видеоиграм.

Компыотер — это просто хранилище битов информации и, устанавливая их в 1 или 0, вы создаете образ. В этом заключается искусство. Я хочу, чтобы вы настроились на созидательную работу. Нам потребуется полное взаимопонимание. В этой главе я расскажу о том, как создаются видеоигры. Вы узнаете вот о чем:

- Кто пишет видеоигры;
- Откуда берутся идеи;
- Фазы создания видеоигры;
- Что вы узнаете из этой книги.

  В следующих главах вы узнаете, как писать игры.

Кто пишет видеоигры?

Видеоигры создаются группами совершенно разных людей. Нас объединяет желание сделать нечто, что заставляет смеяться, улыбаться, в азарте подпрыгивать на стуле. Если вы хотите доставить другим радость, то можете не сомневаться, что видеоигры - это то, что надо! Кроме того, создание видеоигр и нас самих делает счастливыми. Я не думаю, что написание компилятора может сделать кого-нибудь счастливым человеком.

Видеоигры - это способ выражения самых фантастических идей и образов. Многим просто необходима отдушина для воплощения своих безумных фантазий. Кстати, я думаю, что именно поэтому существует театр и кино. Мы верим в свои иллюзии, порой убегая в них от ужасной реальности. Видеоигры — это такой же способ позволить людям хотя бы на время стать богами своих маленьких вселенных.

Откуда берутся идеи?

Идеи видеоигр берутся из нашего воображения. Именно в нашем сознании существует бесчисленное количество миров, населенных роботами, и городов, наводненных призраками. Попробуйте покопаться у себя в голове и найти какой-нибудь сюжет. Если ничего не получится — не расстраивайтесь. Сходите в ближайший прокат видео и возьмите несколько фантастических лент — может быть они помогут разыграться вашей фантазии? Единственное, от чего я хотел бы вас предостеречь — это от переделки чужих игр. Во-первых, это нехорошо, а во-вторых, у вас могут просто появиться серьезные неприятности. В конце концов, ваши сны могут подсказать самые фантастические сюжеты для игр.

Когда у вас, наконец, появится идея, то очень важно дать ей "отстояться". Попробуйте в течение недели или двух просто выкристаллизовать ее у себя в голове. Пусть она станет для вас совершенно ясной, попробуйте играть в нее в своем воображении. Если это произойдет, то считайте, что самая сложная часть уже позади. Осталось самое простое - запрограммировать ваши идеи.

Фазы создания видеоигр

Видеоигра, как и любой другой программный продукт, должна создаваться по определенной методике. Это значит, что мы в процессе разработки должны придерживаться определенных правил и рекомендаций. Итак:

- Во-первых, нужна идея. Мы уже об этом говорили;
- Если есть понимание того, что будет в игре, то есть смысл написать что-то типа сценария. Если игра будет развиваться на нескольких уровнях — опишите каждый из них;
- Затем вам надо разнообразить каждый из уровней какими-нибудь неожиданными ходами, целями и т. д. Вы должны заинтересовать игрока, заставить его проходить уровень за уровнем в вашей игре;
- Если у вас есть понимание каждого уровня игры, то имеет смысл подумать о структуре самой игры. Как будут вести себя игровые объекты, как они будут взаимодействовать, какие возможности получит игрок?

В этот момент у вас уже есть достаточно информации, чтобы садиться и начинать писать более развернутый план игры. Теперь попробуйте чуть более заострить свое внимание на

специфике игры. Например:

- Выберите, в каком видеорежиме у вас будет работать игра. Например, она может быть выполнена в режиме высокого разрешения, но использовать при этом только несколько цветов.
- Подумайте, насколько сложной будет графика. Будет ли она трехмерной или двухмерной.

О том, как решать эти проблемы вы также узнаете из данной книги.

Когда вы решите для себя эти вопросы, настанет время подумать о тех средствах созидания, которыми мы располагаем. Попробуйте начать конструировать с максимальной детализацией самый первый уровень. У вас сразу появится необходимость в специальных инструментальных средствах. Вот их минимальный набор:

- Программа для рисования битовых образов;
- Программа для анимации битовых образов;
- Си-код для бит-блиттинга (блокового перемещения битовых образов), изменения видимого размера объектов (масштабирования) и рисования линий;
- Алгоритмы искусственного интеллекта для персонажей игры;
- Средства для работы со звуком;
- Си-код для работы с устройствами, ввода;
- Инструменты для рисования уровней и сохранения их на диске;
- Наборы MIDI-звуков для каждого из уровней.
- Когда вы начнете писать программу, старайтесь разбить ее на маленькие секции. На самом деле программа может быть разбита на следующие куски:
- Игровой мир и описывающие его структуры данных;
- Система рендеринга;
- Система ввода/вывода;
- Система искусственного интеллекта;
- Основной игровой цикл; Интерфейс пользователя;
- Система звука.
- Система искусственного интеллекта;

Что вы узнаете из этой книги?

Эта книга написана, чтобы научить читателя создавать трехмерные видеоигры типа DOOM или Wolfenstein 3-D. Эту книгу не стоит рассматривать как учебник по видеографике. Книга написана так, что в ней освещаются вопросы, связанные именно с играми и разработкой игр.

Для того чтобы научиться писать игры, нам придется очень серьезно потрудиться, но я уверен, что когда вы закончите работу над этой книгой, то сможете самостоятельно разработать серьезную игру. Чтобы совсем быть в этом уверенным, мы с вами начнем писать игру. Эта такая трехмерная игра типа Wolfenstein 3-D. Впрочем, подойдя к последней главе, вы сами ознакомитесь с тем, что у меня получилось.

ИТОГ

В этой главе мы узнали кое-что о том, как и для чего пишутся видеоигры- Кроме того, выяснилось, что помимо технических навыков для разработки игры нужно иметь:

- Сценарий игры;
- Распределение по ролям;
- Дизайн игры;
- Понимание того, для чего вы все это делаете.

Итак, вроде, мы обсудили все, что находится вокруг игр. Пора начать работать. Давайте начнем.

ОСНОВЫ ЯЗЫКА АССЕМБЛЕРА

Доктор Руди Раккер, мой друг и бывший наставник, как-то сказал: «Ассемблер - это просто круто». Я думаю, эти слова как нельзя лучше подходят к данной главе. Ассемблер — это родной язык всех компьютеров, и если вы им хорошо овладеете, он предоставит вам фантастические возможности. В настоящей главе мы кратко ознакомимся с этим языком.

Мы научимся подключать фрагменты, написанные на ассемблере, к нашим программам и использовать встроенный (in-line) ассемблер компилятора Microsoft C. Кроме того, мы пойдем чуть дальше и напишем еще парочку графических процедур. Таким образом, эту главу можно разделить на следующие части:

- Зачем нам нужен ассемблер при написании игр;
- Описание семейства процессоров 80x86;
- Регистры ЦПУ;
- Общий вид процедуры на ассемблере;
- Передача параметров;
- Локальные переменные;
- Создание внешних ссылок;
- Возвращение параметров;
- Некоторые полезные управляющие конструкции;
- Установка видеорежимов;
- Сверхскоростная очистка экрана;
- Использование встроенного ассемблера.

Зачем нам нужен ассемблер при написании игр?

Даже для таких компьютерных богов, как Microsoft и Borland, сегодня ассемблер намного быстрее программ на Си. Я помню дни, когда все игры были написаны целиком на ассемблере. Вы можете себе это представить? К счастью сейчас у нас есть куча ученых и программистов, которые заняты разработкой компиляторов. Эти компиляторы дают код, не многим хуже ассемблерного. Но, к сожалению, даже компиляторы чуть-чуть не дотягивают до идеала. Вот почему нам обычно приходится последние пять процентов делать своими руками. Мы должны применять ассемблер для рисования точек, линий, выполнения заливок, наложений и т. д. Мы вынуждены его применять там, где нужна сверхскорость - в графике.

Существует несколько способов включения ассемблера в наши программы:

- Можно написать на ассемблере отдельную функцию и вызывать ее из Си. Мы используем этот способ в критичных по времени выполнения участках программы;
- Мы можем использовать такую штуку, как встроенный ассемблер. Он позволяет использовать инструкции ассемблера наравне с Си-кодом. Это неоценимая находка для разработчика видеоигр, поскольку вам не надо использовать кучу промежуточных утилит и создавать массу независимых текстов программ. Кроме того, встроенный ассемблер здорово экономит ваше время.

Прежде чем двинуться дальше, я хочу предостеречь вас от одной из крайностей - не надо пытаться написать на ассемблере все. Используйте его экономно и только тогда, когда он действительно нужен. В противном случае ваш код будет довольно сложен и непереносим (кстати, в том же самом DOOM'e на ассемблере написано всего несколько процедур, а все остальное — это эффективный Си-код). Если же вам надо написать более двух тысяч строк ассемблера, то лучшим решением будет пересмотр применяемого вашего алгоритма.

Обзор семейства процессоров 80x86

Семейство процессоров 80x86 более чем разнообразно. Фирма Intel создала процессоров больше, чем вы в состоянии представить. Игры и программы, которые мы будем создавать в этой книге, ориентированы только на 386-, 486- и 586-процессоры. Для наших целей мы будем говорить о реальном режиме работы этих процессоров: режим эмуляции процессора 8086, используемый DOS с 640-килобайтным ограничением.

Как вы знаете, первым из этого семейства был 8088. Он стоял на первом ПК. Потом его сменил 8086, но это уже история. Весь мир перешел на DOS которая изначально была ориентирована на процессор 8086. Производя процессоры 286, 386, 486, 586 (и вроде скоро появится 686), Intel обеспечивал поддержку 8086-процессора. Это значит, что даже на 586 ПК вы неизбежно натолкнетесь в DOS'e на 640-килобайтный барьер.

На самом деле, это ужасно, поскольку 386 и все последующие 32-разрядные процессоры имеют непрерывную модель памяти (без сегментов), более мощные инструкции и т. д. Использование только новых инструкций сделает наши игры более мощными. Более того,

непрерывная модель памяти устранит понятие сегментных регистров, 640-килобайтного барьера и всего, что с ним связано. Мы можем указать компилятору порождать 286-инструкции, дав соответствующую директиву, но это будут лишь 16-битные команды, которые не дадут нам возможности использования всех средств 32-разрядных процессоров типа 386 486 и 586.

Кстати, для удобства я буду называть 386-, 486- и 586-процессоры просто «процессорами».

В функции процессора входит перемещение данных в памяти и выполнение с ними некоторых преобразований. К ним относятся математические и логические (например, И, ИЛИ, НЕ) операции, проверка различных условий и т. п. Любой процессор должен иметь достаточно много регистров, чтобы его можно было эффективно программировать (они нужны для сохранения данных). Регистры процессора ПК будут описаны в следующем разделе.

Регистры процессора

Регистры общего назначения. Данные регистры используются во время выполнения программ и во многих случая являются взаимозаменяемыми. Кроме того, каждый из них имеет определенное предназначение.

AX - 16 бит, общего назначения, часто именуется аккумулятором;

BX - 16 бит, общего назначения и индексный;

CX - 16 бит, общего назначения и счетчик;

DX - 16 бит, общего назначения;

BP - 16 бит, общего назначения, используется для хранения смещения и индексов, часто называется регистром базы;

SI - 16 бит, общего назначения, используется в операциях с памятью (SI — source issue — регистр источника, используется для хранения смещения Источника при выполнении строковых команд);

DI - 16 бит, общего назначения, используется в операциях с памятью (DI - destination issue — регистр приемника, используется для хранения смещения пункта назначения при выполнении строковых команд).

Сегментные регистры. Данные регистры используются как указатели на сегменты. Сегмент - это блок размером в 64K, который предназначен для определенных целей: для хранения программного кода, данных и т. д.

DS - сегмент данных;

CS - сегмент кода;

ES - дополнительный сегмент;

SS - сегмент стека;

IP - счетчик.

Флаговый регистр

Этот регистр сохраняет статусы состояния процессора, такие как: Z (zero - ноль), C (carry - перенос) и т. д. Этот регистр не доступен напрямую, но его содержимое можно узнать с помощью соответствующих инструкций.

Общий вид ассемблерной функции

Процедуры ассемблера очень похожи на Си-функции. У них есть начало, конец и код, расположенный в середине. В этой книге мы будем ссылаться на Microsoft macro assembler версий от 5.1 до 6.1 (MASM). Это связано с тем, что.в них уже есть директивы, упрощающие стыковку с программами на Си.

Примечание

MASM версии 5.0 подойдет для первого примера, но для успешной работы с книгой вам понадобится версия 5.1 или старше.

Прежде чем начать изучение способов передачи параметров и прочих интересных вещей, давайте посмотрим, что нам нужно как минимум для написания внешней программы на ассемблере, которую мы потом вызовем из Си. Итак, Листинг 2.1 покажет вам прототип процедуры на ассемблере.

Листинг 2.1. Прототип процедуры для MASM 5.0 и более старших версий.

```
.MODEL MEDIUM                ;тип модели памяти
.CODE
; начало кода
PUBLIC _function_name      ;информация для компоновщика. Функция
;может экспортироваться
_function_name PROC FAR      ;название и тип функции(ближняя
;или дальняя). Ближние функции можно использовать для моделей
;памяти SMALL и COMPACT, а дальние применяются для моделей
;памяти MEDIUM, LARGE и HUGE.
push BP                      ;готовим фрейм стека—пролог
;функции
mov BP,SP                    ;сохраним стек
;Работа функции
pop BP                       ;восстанавливаем фрейм стека
;эпилог функции
_function_name ENDP          ; конец процедуры
END
;конец кода
```

Давайте проанализируем программу, приведенную в Листинге 2.1.

- Первая директива, которую мы встречаем — это .MODEL. Как и компилятор Си, MASM должен знать, какая из моделей памяти используется. Ключевое слово MEDIUM означает, что мы собираемся использовать модель памяти именно типа MEDIUM. Теперь я хочу напомнить вам свойства основных моделей памяти:
  - Модель SMALL имеет один 64-килобайтный сегмент для кода и один сегмент для данных;
  - Модель COMPACT имеет один 64-килобайтный сегмент для кода и несколько сегментов данных;
  - Модель MEDIUM имеет один 64-килобайтный сегмент для данных и несколько сегментов для кода;
  - Модель LARGE имеет несколько сегментов как для кода, так и для данных;
  - Модель HUGE разрешает данным быть больше, чем 64K,но в остальном полностью похожа на модель LARGE.

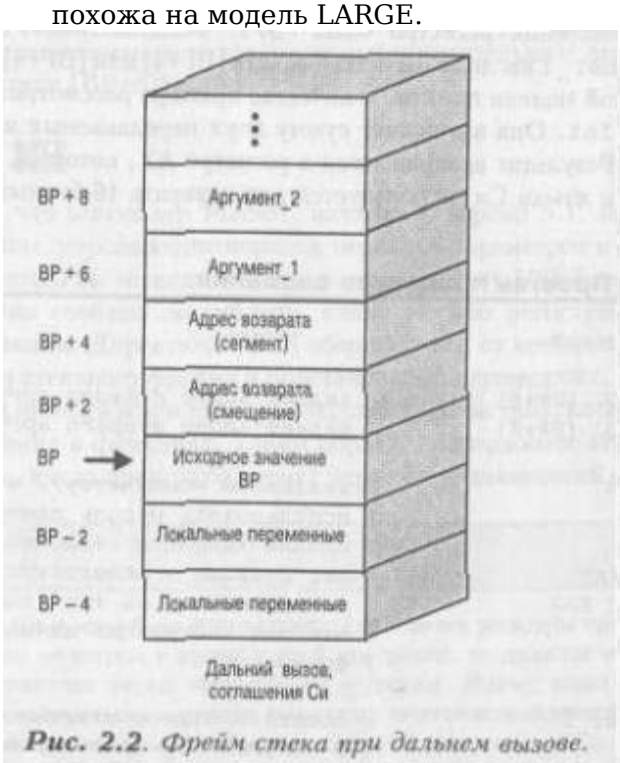


Рис. 2.2. Фрейм стека при дальнем вызове.

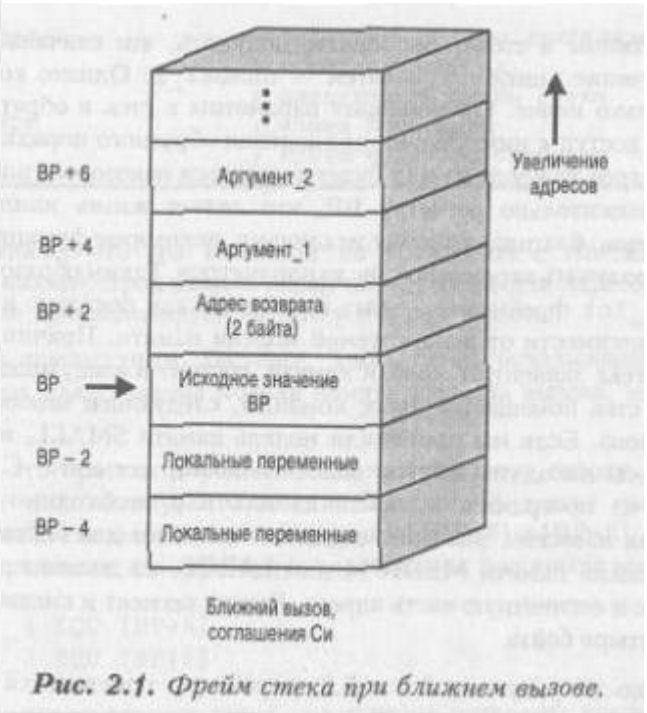


Рис. 2.1. Фрейм стека при ближнем вызове.

Чаще всего мы будем использовать модели памяти MEDIUM и LARGE.

- Следующая директива — PUBLIC. Она говорит MASM, что следующее имя будет экспортировано, то есть станет «видимо» из других модулей;

- Теперь мы переходим к началу самой функции. В ассемблере функция начинается с директивы PROC, которая следует сразу за именем функции;
- В этом месте мы находимся внутри исполняемой части кода. Первые две инструкции устанавливают стек таким образом, что процедура получает доступ к параметрам, передаваемым через стек. К этому мы еще не раз вернемся;
- В конце процедуры мы очищаем стек;
- В конце каждой процедуры ставится ключевое слово ENDP;
- В одном блоке мы можем иметь сколько угодно процедур, но надо помнить, что самой последней должна быть директива END. Она сообщает ассемблеру об окончании программы.

Тема следующего разговора - это передача параметров. Наши ассемблерные функции не были бы так полезны, если б мы не имели возможности обмениваться с ними информацией,

Передача параметров

Языки Си и ассемблер похожи на дальних родственников, живущих в одном доме - они вынуждены придерживаться сложных взаимных условностей. Однако ассемблер значительно более примитивен. Поэтому при передаче параметров ассемблерной процедуре нам приходится сочинять множество дополнительных строк кода, обеспечивающих доступ к ним. Вначале необходимо оформить фрейм стека, как показано в Листинге 2.1. Далее необходимо получить доступ к переданным параметрам, основываясь на новом значении регистра базы (BP). Для обеспечения доступа к параметрам вы должны четко представлять себе, как именно передаваемые параметры размещаются в стеке. К примеру, вы хотите написать процедуру, вычисляющую сумму двух чисел и возвращающую результат в регистре AX. На языке Си, описание этой функции выглядит так:

```
int Add_Int(int number_1, int number_2);
```

При выполнении этой процедуры компилятор языка Си создаст фрейм стека и поместит туда параметры. Иными словами, значения number_1 и number_2 будут расположены в стеке. Вы можете подумать, что сначала в стек будет помещено значение number 1, а затем - number_2. Однако компилятор Си думает несколько иначе. Он помещает параметры в стек в обратном порядке, что облегчает доступ к ним. За счет применения обратного порядка размещения параметров, адрес каждого из них будет задаваться некоторым положительным смещением относительно регистра BP, что делает жизнь намного легче. В частности, именно благодаря такому механизму, некоторые функции (например, printf) могут получать переменное число параметров. Таким образом, при вызове функции Add_Int фрейм стека будет выглядеть, как показано па рисунке 2.1 или 2.2, в зависимости от используемой модели памяти. Причина, по которой вид фрейма стека зависит от модели памяти, состоит в следующем: при вызове процедуры в стек помещается адрес команды, следующей непосредственно за командой вызова. Если мы применили модель памяти SMALL, все процедуры по определению находятся внутри одного кодового сегмента. Следовательно, для доступа из программы к любой из них нам необходимо знать только смещение. Как известно, значение смещения занимает два байта. Если же мы применяем модель памяти MEDIUM или LARGE, то должны сохранить как смещение, так и сегментную часть адреса. Вместе сегмент и смещение занимают уже целых четыре байта.

Как видно из рисунков 2.1 и 2.2, параметры помещаются в стек в том порядке, который обеспечивает их адресацию положительными смещениями относительно значения регистра базы (BP). Следовательно, для доступа к параметру number 1 вы должны использовать [BP+4] или [BP+6], в зависимости от установленной модели памяти. В качестве примера рассмотрим полный текст функции Add_Int. Она вычисляет сумму двух передаваемых в качестве аргументов чисел. Результат возвращается в регистре AX, который, в соответствии с соглашениями языка Си, используется для возврата 16-битных значений.

Листинг 2.2. Простая процедура сложения.

```
; Секция констант
integer_1 EQU [BP+6]        ; задает адрес первого аргумента
integer_2 EQU [BP+8]        ; задает адрес второго аргумента
.MODEL MEDIUM                ; указываем компилятору, что он должен
; использовать модель памяти MEDIUM
.CODE                        ; начало кодового сегмента
PUBLIC _Add_Int              ; эта функция - общедоступна
_Add_Int PROC FAR            ; имя функции и ее тип (дальняя)
push BP                      ; эти две инструкции инициализируют
; фрейм стека
mov BP, SP
mov AX,integer_1             ; помещаем первое слагаемое
; в аккумулятор (регистр AX)
add AX,integer_2             ; добавляем второе, слагаемое
; к содержимому AX
pop BP                       ; ликвидируем фрейм стека
_Add_Int  ENDP               ; конец процедуры
END                          ; конец кодового сегмента
```

Единственное, что мы изменили по сравнению с Листингом 2.1, это добавили несколько

строк кода и ввели определения для адресов параметров. Теперь давайте проанализируем то, что у нас получилось.

- Как и в предыдущем листинге, здесь были использованы директивы ассемблера для указания модели памяти, способа вызова, начала и конца функции;
- EQU — это простая директива, заменяющая одну строку на другую. Я прибег к ней потому, что мне не хотелось в тексте самой функций использовать синтаксические конструкции [BP+6] и [BP+8]. Строки, задающие выражения, которые будут подставлены при компиляции, это:

```
integer_l EQU [BP+6]

integer_2 EQU [BP+8]
```

В общем, использование таких подстановок позволяет сделать ассемблерную программу более читабельной. Единственной альтернативой такому подходу является написание команды индексирования относительно содержимого одного из регистров (типа [BP+6]).

Директива USES

Надо сказать, что ассемблер MASM, начиная с версии 5.1, имеет некоторые новые директивы, упрощающие порядок передачи параметров и создания фрейма стека. Для этого вы можете использовать директиву USES вместе с директивой PROC. Они сообщат ассемблеру, какие именно регистры будут использоваться в функции. Директива USES оберегает вас от всей рутины, связанной с определением стекового фрейма и подстановками переменных. Более того, она генерирует код пролога и эпилога для сохранения регистров, которые вы указали для использования в функциях. Таким образом, содержимое этих регистров не будет изменено, когда процедура вернет управление вызвавшей ее Си-функции.

Внимание!

Помните, что Си и ассемблер используют одни и те же регистры процессора. Если вы пользуетесь регистром в ассемблерной программе, то должны его сохранить в стеке и восстановить перед завершением функции. Иначе, ваша Си-программа может просто "сломаться" в момент выхода из вызова ассемблерной вставки.

Директива PROC и относящийся к ней уточнитель USES имеет следующий синтаксис.

```
label PROC [[attributes]] [[USES register_list]] [[,]]
                          [[parameter list][:type]]...]]
```

- Поле label — это имя процедуры;
- Поле attributes сообщает ассемблеру свойства вашей процедуры. Она может содержать множество параметров, таких как тип процедуры (NEAR или FAR), «видимость» процедуры (PUBLIC или PRIVATE) и, наконец, тип языка (C, PASCAL и т. д.). Эта возможность делает наши программы на ассемблере более читаемыми. Правда, это связывает руки, но зато программы обретают определенную элегантность;
- Поле register_list показывает, какие регистры будет использовать функция. При этом ассемблер генерирует код, который может сохранить их на время работы процедуры и восстановить при выходе;
- Поле parameter_list очень похоже на список параметров в Си;

Для каждой передаваемой процедуре переменной должен быть указан тип, определяющий их размер (например, BYTE или WORD). Тип задается в поле type.

Если вы пишите процедуру, в которую передаете три целых величины, и будете использовать регистры SI, DI и CX, то должны включить следующий оператор:

```
far proc USES SI DI CX, integer_1:WORD, integer_2:WORD,
                        integer_3:WORD
```

Используя директивы PROC и USES, давайте перепишем процедуру из Листинга 2.2.

Листинг 2.3. Модифицированная версия Add_Int.

```
.MODEL MEDIUM,C              ; использовать модель MEDIUM ; и соглашения по вызову Си
.CODE                        ; начало кода

PUBLIC _Add_Int             ; объявляем функцию как общедоступную
_Add_lnt PROC USES integer_1 :WORD, integer_2 :WORD
mov AX,integer_l            ; загрузить первый операнд в AX
add AX,integer_2       ; сложить второй операнд с AX
_Add_Int ENDP          ; конец процедуры
END                    ; конец кода
```

Как видно из Листинга 2.3, тяжкое бремя сохранения регистра BP, создания и уничтожения стекового фрейма теперь отдано на откуп ассемблеру. Более того мы получили прямой доступ к параметрам integer 1 и integer 2.

Передача указателей

Мы знаем, как передать значения таких параметров как BYTE или WORD, но как передать указатель? Указатели передаются как двойные слова, или DWORD. Для доступа к указателям в стеке нам придется воспользоваться старым приемом: разобьем двойное слово указателя на две переменные segment и offset, которые будут иметь тип WORD, и уже к ним будем обращаться в. нашей ассемблерной программе. К примеру, если мы вызываем ассемблерную функцию в модели MEDIUM, (скажем, это будет вызов типа FAR) в следующей строке:

```
pfoo(&x)
```

то получить адрес переменной X можно будет с помощью следующих подстановок:

```
offset EQU [BP+6] segment EQU [BP+8]
```

Если мы захотим изменить значение X, то нам придется сделать следующее:

```
mov DI,offset

mov AX,segment

mov ES,AX

mov ES:[DI],CX
```

Эта программа состоит из двух основных частей:

- Во-первых, создается указатель на X через регистры ES и DI;
- Во-вторых, изменяется значение переменной X.

Ну вот и все о том, что связано с передачей параметров. Новые расширения директив PROC и USES просто великолепны, и вы можете всегда ими пользоваться, если чувствуете от их применения комфорт. Если вы предпочитаете все делать в стиле MASM 5.0, то это ваше право. С точки зрения быстродействия программы здесь нет никакой разницы.

Локальные переменные

Теперь вы знаете, как передавать переменные в процедуры, а вот как насчет временных и локальных переменных, которые действуют только внутри процедуры?

Когда мы пишем программы на Си, то применяем локальные переменные для вычислений, сохранения результатов и т. д. Так же, как и в Си, мы можем создать локальные переменные в наших ассемблерных функциях, используя стек. Конечно, можно создать локальные переменные в области данных (например, в сегменте данных), но этого не стоит делать. Почему? Дело в том, что стек как раз и создавался для временного хранения данных и локальных переменных, так что именно им и стоит пользоваться.

Давайте посмотрим, как это делается. Когда мы определяем стек, сохраняя регистр BP, то можем извлекать параметры путем прибавления положительного смещения к регистру BP, например, [BP+6] и т. д. Таким образом, получается, что в действительности стек — это область непрерывной памяти, которую мы можем использовать также и для хранения локальных переменных. Для этого нам надо только использовать отрицательное смещение относительно регистра BP.

В случае, если мы хотим иметь две локальные переменные 1ocal_1 и 1oca1_2, можно использовать следующую подстановку:

```
local_1 EQU [BP-2]

local_2 EQU [BP-4]
```

Это дает нам два целых числа. Пока нам не известно, что записано по этим адресам и мы можем только предполагать, что данный участок памяти можно использовать безболезненно. Однако нужно помнить, что мы только что использовали стек для хранения данных и теперь нам необходимо самим изменить регистр SP для отражения этого. Если этого не сделать, то первая встретившаяся инструкция PUSH обязательно что-нибудь запишет в нашу переменную и непременно «испортит» ее.

Для исключения этой ситуации можно предложить уменьшить значение регистра SP на длину переменных, которые мы хотим сохранить. В нашем случае это составит четыре байта. В Листинге 2.4 показано, как это сделать.

Листинг 2.4. Корректировка регистра SP.

```
push BP         ; устанавливаем фрейм стека, как обычно
mov BP,SP
sub SP,4        ; корректируем указатель стека. Теперь наши
;переменные не будут случайно изменены
;Вся работа выполняется здесь
```

```
add SP,4        ; перед уничтожением фрейма стека надо восстановить
; исходное значение указателя стека
pop BP
; уничтожаем фрейм стека,
; восстанавливая регистр BP
```

Директива LOCAL

Заметьте, что в Листинге 2.4 мы изменили значение регистра SP не только в начале процедуры, но и в конце (перед тем как восстановить регистр BP). Эта техника обычно используется для размещения переменных в ассемблерных процедурах при их вызовах из языков высокого уровня.

В листинге 2.4 это делалось вручную. А вот MASM 5.1 и более поздние версии имеют встроенную директиву, которая выполняет это автоматически. Это директива LOCAL и она имеет следующий синтаксис:

```
LOCAL variable name: type, variable name: type, ...
```

(Любопытно. MASM все больше становится похож на Си. К чему бы это?) Давайте теперь напишем программу с использованием директивы LOCAL. Она называется Timer и требует одного параметра — time, который затем помещает в локальную переменную asm time. Из Си этот вызов будет выглядеть так:

```
Timer(25);
```

Листинг 2.5 показывает реализацию программы Timer на ассемблере, используя все директивы, которые мы обсудили в этой главе.

Листинг 2.5. Программа Timer.

```
.MODEL MEDIUM                ;используем модель MEDIUM
.CODE                        ;начало кодового сегмента
;в процессе работы функция меняет содержимое регистра AX
_Timer PROC FAR USES AX, time:WORD   LOCAL asmt_time :WORD
mov AX, time
mov asm_time, AX
_Timer ENDP END
```

Эта программа оказалась бы куда длиннее, если б мы не использовали новые директивы MASM. Правда, если у вас есть только MASM версии 5.0, то вы можете обойтись и без них.

Совет

Я надеюсь, что вы создадите свои шаблоны, позволяющие обращаться к передаваемым параметрам и локальным переменным.

Создание внешних ссылок

Когда вы пишите модуль на Си, в котором встречаются переменные или функции, определенные в других модулях, вы должны использовать ключевое слово EXTERN, сообщающее компилятору, что переменные или функции будут определены позже (па этапе компоновки). MASM 5.0 и более старшие версии также поддерживают эту возможность.

В наших ассемблерных функциях может понадобиться передать значение глобальной переменной обратно в вызывающую Си программу. В принципе, мы можем ее передавать как параметр каждый раз и не думать о внешних переменных. Но данный способ критичен по времени и весьма архаичен. В видеоиграх, как ни в каких других программах, мы должны использовать как можно больше глобальных переменных. Причина очевидна — скорость выполнения. Нет смысла терять драгоценные такты процессора на выполнение команд PUSH и POP в момент вызова функций.

Синтаксис директивы EXTRN следующий:

```
EXTRN symbol: type, symbol: type,...
```

где symbol — имя переменной, а type — ее размер (например, BYTE, WORD, DWORD).

Директива EXTRN разрешает разместить переменную в вашем Си-коде и получить к ней доступ через параметры. Это имеет и обратную сторону: переменная, обозначенная как EXTRN означает, что она занимает текущий сегмент данных, адресуемых через регистр DS. Если вы все будете делать в модели SMALL или MEDIUM, то не стоит беспокоиться, если же вы работаете в модели LARGE, то никто не гарантирует, что вы получите доступ к вашей глобальной переменной, используя текущее значение регистра DS. Чтобы избежать этого, всегда применяйте модели SMALL и MEDIUM.

Давайте для примера напишем процедуру, которая складывает два целых числа и помещает результат в третье. Фокус заключается в том, что все эти величины будут глобальными и по отношению к ассемблерной процедуре - внешними. Листинг 2.6 демонстрирует код Си для этой программы, а в Листинге 2.7-показана ее реализация на ассемблере.

Листинг 2.6. Си-часть примера.

```
#include <stdio.h>
int first = 1, second = 2, third = 0;
// Это те числа, с которыми
// мы хотим работать
void main (void)
{
      printf ("\nBefore adding third = %d\ third);
      Add_Ext();
      //вызываем ассемблерную процедуру
      printf("\nAfter adding third = %d",third);
} // конец функции main
```

Листинг 2.7. Ассемблерная часть примера.

```
.MODEL MEDIUM                 ; будем использовать MEDIUM модель
EXTRN first:WORD, second:WORD, third:WORD
.CODE                         ; начало кодового сегмента
_Add_Ext PROC FAR             ;процедура имеет тип FAR (дальняя)
mov AX, first                 ; помещаем первое число в аккумулятор
add AX, second                ; прибавляем второе число
mov third, AX                 ; помещаем результат в переменную third
_Add_Ext ENDP                 ; конец процедуры
END                           ; конец кодового сегмента
```

Листинги 2.6 и 2.7 - это примеры использования внешних переменных first, second и third. Программа вызывает Add_Ext, которая складывает переменные first и second и сохраняет результат в third. Это подводит нас к теме возврата результатов обратно в программу на Си, которая вызывала ассемблерную процедуру.

Возвращение параметров в вызывающую функции

Когда Си только начинал создаваться, одним из требований в спецификациях называлась «функциональность» языка. Под функциональностью я понимаю возвращение результата и возможность использования комплексных выражений. Например, рассмотрим следующее выражение на Си:

```
coeff = Factorial(n)*cos(r)*Scale(z);
```

Это выражение использует три функции и выполняет требуемые математические операции. Результат сохраняется в переменной coeff. Именно это и делает Си «функциональным» языком.

Конечно, это только одна из многих возможностей Си. Таким образом, если мы хотим эмулировать с помощью ассемблера настоящие Си-функции, мы должны иметь возможность возвратить результат в вызывающую функцию, как это делает Си.

Если вы скомпилируете парочку Си-функций, возвращающих значение, и посмотрите на их ассемблерные листинги, то вскоре заметите, что функции всегда возвращают результат в строго определенном наборе регистров. Если же вы прислушаетесь к моим советам, то я гарантирую, что ваши ассемблерные функции будут, по крайней мере, правильно возвращать результаты.

В зависимости от типа, возвращаемые в Си параметры должны находиться в следующих регистрах:

- BYTE возвращается в регистре AL;
- WORD возвращается в регистре AX;
- DWORD должно возвращаться ,в, паре DX:AX, причем в AX записывается
- младшее слово;
- Указатели типа NEAR должны возвращаться в регистре AX;
- Указатели типа FAR Возвращаются в паре DX:AX, причем в DX должен содержаться сегмент, а в AX - смещение.

Давайте для примера вспомним Листинг 2.2, где мы складывали два целых числа и накапливали результат в AX. К счастью, это именно тот регистр, в котором целое значение может быть возвращено в Си-программу. Если же встречается другая ситуация, (например, результат находится в регистре CX). то для корректной передачи результата мы должны перед выходом переместить полученное значение в AX.

Так... Вроде бы, с директивами и техникой программирования на ассемблере MASM мы покончили. Теперь я предлагаю перейти к более живым примерам. Кстати, для них-то все это и написано.

Установка видеорежимов

В играх есть немало технических хитростей: работа со звуком, «искусственный интеллект» и многое другое. Но перед тем как начать этим заниматься, давайте попробуем инициализировать наш дисплей.

Конечно, мы сразу можем набрать гору документации с подробным описанием устройства дисплея, регистров и установок, но все это весьма опасно, и вот почему. То, что будет работать на одной видеокарте, может оказаться абсолютно неработоспособным на другой. Таким образом, чтобы избежать возможной несовместимости, для установки видеорежима мы будем использовать базовую систему ввода/вывода (BIOS).

Можно смело сказать, что основная графическая мощь ПК сосредоточена в прерывании с номером 10h. Использование этого прерывания весьма просто — необходимо правильно установить нужные регистры процессора в зависимости от выполняемой функции. В этой книге мы будем пользоваться режимом 13h (это графический режим с разрешением 320x200 точек, при 256 цветах). Теперь нам нужно найти, как перенести компьютер в этот режим. Для этого давайте напишем программу на ассемблере для установки режима 13h и программу на Си для проверки. Соответствующие фрагменты показаны в Листингах 2.8 и 2.9.

Листинг 2.8. Ассемблерная процедура, устанавливающая видеорежим (SETMODEA.ASM).

```
.MODEL MEDIUM, C
;модель памяти - MEDIUM, соглашения языка Си
.CODE                   ;начало кодового сегмента
PUBLIC Set_Mode         ;объявляем функцию как общедоступную
Set_Mode PROC FAR C vmode:WORD ;функция получает один параметр
mov AH,0                ;функция 0 прерывания 10h - установка режима
mov AL,BYTE PTR vmode ;номер режима, который вы хотите установить
int 10h                 ; используем BIOS для установки режима
ret                     ; возврат из процедуры
Set_Mode ENDP           ; конец процедуры
END                     ;конец кодового сегмента
```

Листинг 2.9. Си-функция, тестирующая видеорежим (SETMOPEC.C).

```
#include <stdio.h>
#define VGA256 0x13
#define TEXT_MODE  0x03
extern Set_Mode(int mode);
void main(void)
{
// устанавливаем режим 320x200 точек, 256 цветов
Set_Mode(VGA256);
// ждем нажатия любой клавиши
while (kbhit()) {}
// возвращаем компьютер в текстовый режим
Set_Mode(TEXT_MODE);
} // конец функции main
```

Теперь если вы наберете и запустите эти программы, то, скорее всего, увидите пустой экран. Правда, в данном случае это не означает «зависание» компьютера, а свидетельствует о том, что VGA-карта переключилась в режим 13h. Стоит только нажать любую клавишу, и вы вновь окажетесь в привычном текстовом режиме 80x25. Конечно, можно было бы использовать функцию _setvideomode() из графической библиотеки Microsoft C, но наша функция работает в сотни раз быстрее.

Теперь, когда мы научились переключать экран в графический режим, неплохо бы попробовать его очистить.

Сверхскоростная очистка экрана

Экран в режиме 13h отображается в области памяти, начиная с адреса A000:0000 и по A000:FBFF. При этом каждый пиксель задается одним байтом. Давайте посмотрим на рисунок 2.3, чтобы лучше понять, как это происходит. В данной конфигурации каждый пиксель может принимать одно из 256 значений, но эти значения не являются, как можно было бы подумать, кодом цвета пикселя. Они представляют собой *видимое значение цвета.*

Теперь поговорим о том, как это работает. Значение пикселя, а это байт, адрес которого определяет положение пикселя на экране, используется в качестве индекса в гигантской таблице цветов. Таким образом, значение пикселя 26 не означает цвет номер 26. Наоборот, это значит «индекс, указывающий на 26-й элемент таблицы и использующий значение этого поля».

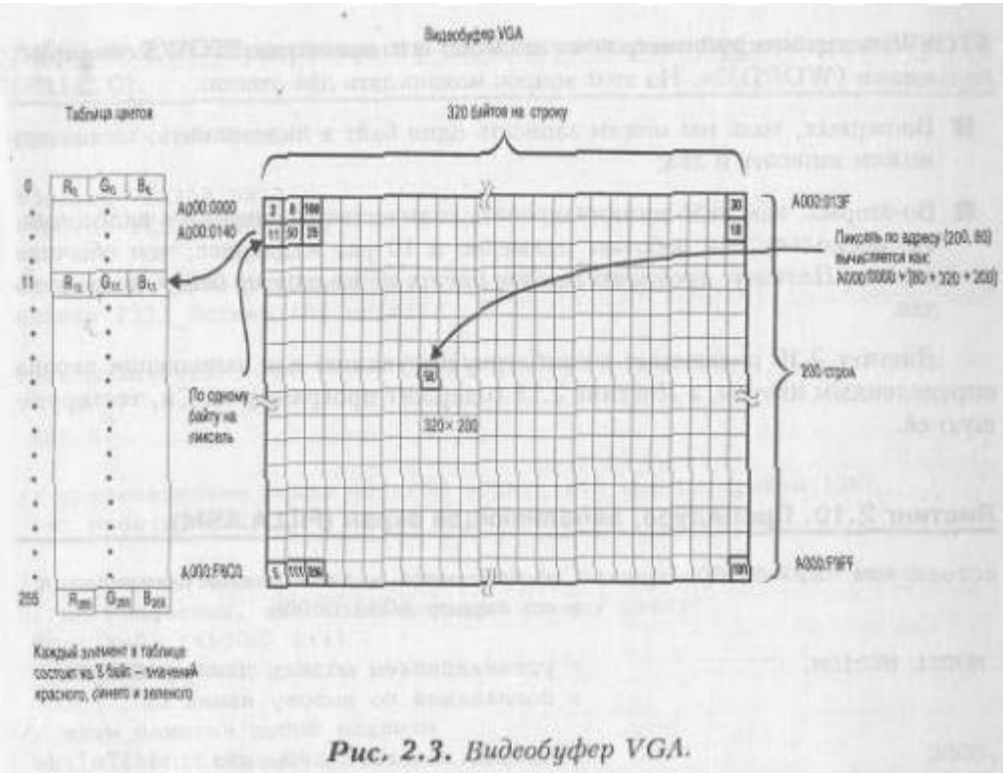


Рис. 2.3. Видеобуфер VGA.

Примечание

Таблица цветов содержит по одному байту для каждого из первичных цветов, Однако реально используются только шесть первых бит каждого байта. Таким образом, каждый из элементов таблицы состоит из трех байтов, определяющих значения трех основных цветов; красного (R - Red), зеленого (G - Green) и голубого (B — Blue), которые в сумме позволяют представить 262114 цветов. Однако размер таблицы ограничен 256-ю элементами, поэтому и на экране может одновременно присутствовать не более 256 цветов.

В видеоиграх, которые нас привлекают, экран перерисовывается от 15 до 30 раз в секунду. Таким образом, перед тем как нарисовать что-то новое на экране, нам необходимо удалить старое изображение. Для того чтобы это делать, надо найти способ быстрого заполнения видеобуфера каким-нибудь значением цвета, например, цветом фона.

Это значение будет заполнять всю видеопамять, а, следовательно, и весь экран в виде цвета, Самый быстрый способ сделать это - воспользоваться ассемблерной инструкцией STOSW. Вы можете спросить: «А зачем использовать STOSW, когда каждый пиксель — это байт, в то время как STOWS оперирует со словами (WORD)?". На этот вопрос можно дать два ответа:

- Во-первых, коль мы можем записать один байт в видеопамять, то значит, можем записать и два;
- Во-вторых, нам надо минимизировать количество обращений к видеопамяти, поскольку она работает примерно в 10 раз медленнее, чем обычная память. Поэтому, предпочтительнее писать не по одному байту, а сразу по два.

Листинг 2.10 показывает ассемблерную функцию для заполнения экрана определенным цветом, а Листинг 2.11 содержит программу на Си, тестирующую ее.

Листинг 2.10. Процедура, заполняющая экран (FILLA.ASM).

```
screen_ram EQU 0A000h       ; видеопамять в этом режиме начинается
; по адресу A000:0000h
.MODEL MEDIUM, C            ;устанавливаем модель памяти MEDIUM,
; соглашения по вызову языка Си
.CODE                       ; начало кодового сегмента
PUBLIC Fill_Screen          ; объявляем процедуру общедоступной,
Fill_Screen PROC FAR C color : WORD ;функция принимает один параметр
mov AX, screen_ram           ;ES:DI должно указывать на видеопамять
mov ES,AX
xor di,di                    ;обнуляем DI
mov CX,320*200/2
;количество слов, которое надо вывести
mov AL,BYTE PTR color        ;помещаем в регистр AL код цвета
mov AH,AL                    ; этот же код помещаем в регистр AH
rep STOSW                    ;эта команда заполняет видеопамять
; выбранным цветом с максимально
; возможной скоростью
RET                          ;выход из процедуры
Fill_Screen ENDP             ;конец процедуры
END                          ;конец кодового сегмента
```

Листинг 2.11, Программа на Си для тестирования программы 2.10 (FILLC.C).

```
#inciude <stdio.h>
#define VGA256 0x13
```

```
#define TEXT_MODE 0x03
extern Set_Mode(int mode);
extern Fill_Screen(int color);
void main(void)
{
int i;
// устанавливаем режим 320x200 точек, 256 цветов (режим 13h)
Set_Mode(VGA256);
// заполняем экран цветом с кодом 1 (в,палитре, устанавливаемой
// по умолчанию, это соответствует синему цвету)
for (t=0; t<1000; t++) Fill_Screen(1) ;
// ждем нажатия любой клавиши
while(!kbhit()) {}
// возвращаемся в текстовый режим работы экрана
Set_Mode(TEXT_MODE);
} // конец функции main
```

Эти программы чистят экран с максимальной скоростью.

Примечание

Я произвел замер скорости работы этих функций на своем компьютере и получил значение 22 кадра в секунду. Это представляется невероятно медленным, и я сильно забеспокоился. Однако при ближайшем рассмотрении выяснилось, что причина задержки — крайне низкое быстродействие видеопамяти. Собственно, процессор мог бы обеспечить скорость до 250 кадров в секунду. Однако, увы, он часто вынужден ждать, пока видеопамять соизволит откликнуться на его обращение.

На прилагаемой к этой книге дискете вы найдете программу под названием GAUGE.EXE, Вы можете использовать ее для замера производительности вашей видеосистемы.

Наш курс ассемблера проходит отлично. Я уже сам узнал кучу нового и надеюсь, что вы тоже. Теперь нам осталось узнать еще про одну возможность программирования на ассемблере: об использовании встроенного (in-line) ассемблера,

Использование встроенного (in-line) ассемблера

Текущая тенденция компьютерной индустрии — использование одного языка программирования. Встроенный ассемблер является откликом на это пожелание. Это платформно-ориентированное расширение стандарта Си. Оно позволяет программистам включать инструкции ассемблера прямо в их программы, а не создавать самостоятельные процедуры и транслировать их отдельно. Это имеет два преимущества:

- Во-первых, программисту на Си теперь не нужно знать так много обо всем, как прежде;
- Во-вторых, встроенный ассемблер — это максимальное удобство в программировании на разных языках.

Встроенный ассемблер объявляется следующим образом:

```
_asm{
инструкции ассемблера;
}
```

Это все, что надо: ни директив, ни пролога, ни эпилога. Только одно ключевое слово _asm, да пара скобок - и мы у цели.

В связи с использованием встроенного ассемблера на ум сразу приходит миллион вопросов. Я попытаюсь ответить, по крайней мере, на половину из них. Шутка. Кстати, есть пара очень важных моментов:

- Не надо заботиться о сохранении регистров — это сделают за вас;
- Передача параметров очень упрощена. К переменным, находящимся в области видимости функции, содержащей ассемблерный код, вы вполне можете из ассемблерного блока обратиться по имени так же, как и из любого другого блока функции.

Например, если мы хотим обменять два значения с помощью встроенного ассемблера, то следующий Си-код показывает, как это сделать:

```
void Swap(int num_1, int num_2)
{
_asm{
mov AX,num 1
mov BX,num 2
```

```
mov num 1,BX
mov num_2,AX
}
}
```

После того как эта процедура произведет обмен, содержимое рабочих регистров окажется полностью разрушенным. Но об этом не стоит беспокоиться, так как компилятор сам позаботиться об их сохранении.

Для большинства задач возможностей встроенного ассемблера вполне достаточно. Конечно, список его функциональных возможностей намного меньше, чем у MASM, однако в большинстве случаев их вполне хватает.

К специалистам в написании компьютерных игр часто обращаются с вопросом: «А насколько быстро это работает?» Так вот, встроенный ассемблер работает практически так же быстро, как и «настоящий». В большинстве случаев вы не увидите абсолютно никакой разницы. Единственное, в чем встроенный ассемблер уступает MASM'у, так это в скорости входа и выхода из функции, поскольку он содержит обязательные для Си-функций пролог и эпилог.

Тем не менее, я призываю вас использовать встроенный ассемблер. Например, поддержка звуковых карт написана мною именно на встроенном ассемблере и мне на это понадобилось всего несколько часов. При использовании настоящего ассемблера мне пришлось бы потратить примерно в два раза больше времени за счет использования отдельных программ для компиляции, линковки и отладки. Что во встроенном ассемблере действительно ужасно, так это ограничения, накладываемые на совместимость с другими компьютерными платформами. Однако подумаем, так ли это страшно, если мы решили писать программы именно для IBM?

И, наконец, маленькая дополнительная информация. Раньше мы решили, что будем использовать процессоры старших моделей только как быстрый 8086. Однако, если вы хотите, весьма неплохо использовать во встроенном ассемблере инструкции процессора 80286, соответствующим образом изменив настройки компилятора. Конечно, можно также использовать команды процессоров 386, 486, 586. Только прежде подумайте: если вы будете ориентироваться на процессор 586, не окажется ли потенциальный рынок для вашей игры слишком ограниченным...

ИТОГ

Эта глава многому нас научила. За короткое время мы узнали уйму нового:

- Мы изучили архитектуру семейства процессора 80x86;
- Мы узнали, как использовать новые директивы макроассемблера для облегчения написания ассемблерных процедур;
- Мы узнали, как вызывать функции на ассемблере из Си-программ;
- Наконец, мы написали несколько программ, которые переводят компьютер в режим 13h, и посмотрели, как организован их вызов из Си.

Теперь вы стали намного в более близких отношениях с MASM'ОМ и языком ассемблера для ПК вообще. Мы обсудили все основные темы, которые важны при программировании видеоигр, включая интерфейс с языком Си и встроенный ассемблер.

Без сомнения, мне надо было научить вас основам программирования на ассемблере. Вы должны уметь использовать ассемблер для решения разных Задач. В этой книге мы еще не раз с ним столкнемся, и вы должны быть уверены, что не утонете в море единиц и нулей. Если вы считаете, что еще не все поняли, то лучше возьмите и перечитайте какую-нибудь хорошую книгу об ассемблере, а уж потом переходите к изучению следующей главы.

ОСНОВЫ РАБОТЫ С УСТРОЙСТВАМИ ВВОДА

В настоящее время компьютеры еще не научились читать наши мысли, а раз так, то для того, чтобы хоть как-то общаться с ними и «объяснять» им, что же мы от них хотим, нам приходится «разговаривать» с ними через механические интерфейсы. В понятие интерфейс входят клавиатура, джойстик и мышь. Для работы с этими устройствами мы будем придерживаться стандартной тактики - использовать BIOS, прерывания и ассемблер. Эта глава состоит из следующих частей:

- Взаимодействие с пользователем в видеоиграх;
- Джойстик и работа с ним;
- Работа с клавиатурой;
- Управление мышью.

Взаимодействие с пользователем в видеоиграх

Компьютерные игры не были бы так азартны, если бы мы не могли влиять на разворачивающиеся в них события. Основная идея видеоигры состоит в возможности, взаимодействия с «виртуальным миром» в реальном масштабе времени. Когда видеоигры только появились, они имели совершенно убогий интерфейс - многие из них управлялись только парой клавиш.

Но время шло и появлялись новые устройства ввода. Фирма Atari и некоторые другие начали выпускать игры, управляемые джойстиками и более хитрыми устройствами типа «пилотского штурвала». Сегодня для ПК создано множество устройств - плод кропотливых теоретических и практических инженерных разработок.

Как создатели видеоигр, мы должны уделять пристальное внимание работе с устройствами ввода. Люди, играющие в наши игры, должны иметь удобный и надежный интерфейс для изучения созданного нами мира. Если же взаимодействие с системой неудобно и сложно, не поможет ни красивая музыка, ни видеоэффекты — людей не привлечет игра, которой они не смогут управлять. Надо сказать, что сейчас очень удачное время для тех, кто собирается заняться видеоигорным бизнесом. Сегодня мы можем выделить среди пользователей ПК несколько больших групп. Например:

- Большинство имеет компьютеры 386 и 486;
- У большинства есть, как минимум, VGA-карты и многие располагают звуковыми картами;
- Наконец, благодаря президенту Microsoft Билу Гейтсу и большой популярности оболочки Windows, практически у всех есть мыши.

Можно задать вопрос: «А имеет ли предполагаемый игрок джойстик?» Ответ будет менее определенным: «кто-то имеет, а кто-то — нет». Многие все же предпочитают клавиатуру. Неважно, какое из устройств используется, мы должны добиться, чтобы его применение было для игрока почти интуитивным. Не в нашей власти выбирать, какое из устройств должно поддерживаться игрой. Мы обязаны научиться работать с любым из них. В данном случае мы должны стать рабами тех, для кого пишем игры, хотя наша философия и привычки (в выборе устройств ввода) могут сильно отличаться от привычек наших игроков.

Теперь без лишних слов начнем с джойстика.

Джойстик

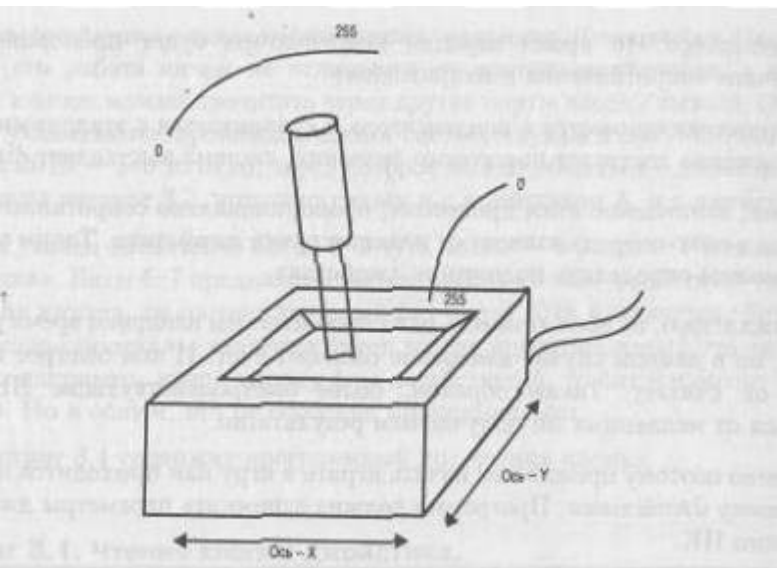


Рис. 3.1. *Динамика механических движений джойстика.*

Одному богу известно, за что джойстик получил столь неуклюжее имя. Интерфейс джойстика с ПК тоже нельзя назвать продуманным, да и аппаратная часть весьма неудобна (правда, здесь не все могут согласиться со мной). Таким образом, по сравнению с другими компьютерами работа с джойстиком на ПК весьма не ортодоксальна и противоречива, но и не так сложна, как может показаться на первый взгляд. Как видно из рисунка 3.1, джойстик - это *аналоговое устройство,* которое изменяет значение сигнала на выходе в зависимости от положения рукоятки.

Наша задача состоит в преобразовании этого аналогового сигнала (его величины) в более приемлемый вид, а именно, в цифровое значение. Все мы, конечно, знаем про АЦП, и если бы ПК создавали сегодня, то он непременно стоял бы в каждой карте порта джойстика.

Но в конце 70-х, начале 80-х годов, когда

ПК только разрабатывались, все АЦП были очень дороги. Тогда инженеры создали специальный АЦП только для контроллера джойстика. Для того времени это было просто гениальным решением, но сегодня заставляет ломать голову над программированием джойстика каждого программиста,

Как работает джойстик

Теперь поговорим более подробно о том, как работает джойстик:

- Каждая ось джойстика имеет связанный с ней потенциометр. Когда рукоятка отклоняется по оси X или Y, то сопротивление соответствующего потенциометра изменяется;
- Потенциометр, используется вместе с конденсатором для создания цепи нагрузки;
- Как мы знаем из курса элементарной электроники, если подать напряжение на цепь, состоящую из последовательно включенного сопротивления и конденсатора, то время зарядки конденсатора будет пропорционально величине сопротивления и напряжения;
- Напряжение снимается с конденсатора и сравнивается с эталонным. Когда напряжение достигает порогового значения, система выставляет флаг;
- Время, занимаемое этим процессом, пропорционально сопротивлению, которое в свою очередь зависит от позиции ручки джойстика. Таким образом, мы можем определить положение джойстика.

К сожалению, во всем этом есть одна проблема: мы измеряем время реакции системы, но в данном случае измерение означает счет. И чем быстрее ПК, тем быстрее он считает. Таким образом, более быстродействующие ПК, будут отличаться от медленных по получаемым результатам.

Именно поэтому прежде чем начать играть в игру нам приходится поиграть в *калибровку джойстика.* Программа должна запомнить параметры джойстика для данного ПК.

Кнопки джойстика

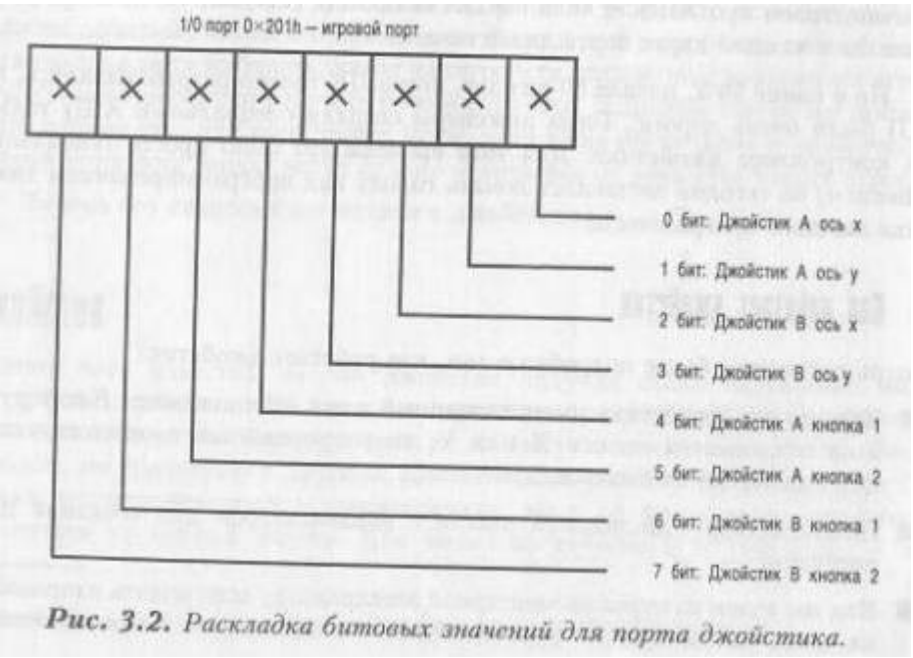


Рис. 3.2. *Раскладка битовых значений для порта джойстика.*

Кнопки джойстика - это обычные переключатели (кстати, вам крупно повезло, ведь это легко могли быть зарядные, ионообменные управляемые магнитным полем подушечки). Для того чтобы узнать состояние переключателя, нам надо прочитать значение из определенного порта.

Есть джойстики с множеством кнопок, например, Trustmaster. Несмотря ни на что, его работа ничем не отличается от других джойстиков, а состояние лишних кнопок можно прочитать через другие порты ввода/вывода. Эти порты обычно указываются производителем в соответствующей документации. Обычно, порт 201h — это то окно, через которое можно общаться с джойстиком. Как показано на рисунке 3.2, этот порт связан и с джойстиком А, и с джойстиком В.

Мы узнаем назначение битов 0-3 чуть позже — в разделе «Чтение позиции джойстика». Биты 4-7 предназначены для чтения кнопок джойстика. Когда одна из кнопок нажата, то соответствующий бит порта 201h изменяется. Есть только одна маленькая деталь: значения битов всегда инвертированы. Это значит, что если вы, например, жмете кнопку 1 на джойстике А, то бит 0 изменит значение с \ на 0. Но в общем, это не особенно принципиально.

Листинг 3.1 содержит программный код чтения кнопки.

Листинг 3.1. Чтение кнопок джойстика.

```
#define JOYPORT      0x201 // порт джойстика = 201h
#define BUTTON_1_A   0x10   // джойстик А, кнопка 1
#define BUTTON_1_B   0x20  // джойстик А, кнопка 2
#define BUTTON_2_A  0x40   // джойстик В, кнопка 1
#define BUTTON_2_B   0x80   // джойстик В, кнопка 2
#define JOYSTICK_1_X 0x01   // джойстик А, ось ,X
```

```
#define JOYSTICK_1_Y 0x02   // джойстик A, ось Y
#define JOYSTICK_2_X 0x04   // джойстик B, ось X
#define JOYSTICK_2_Y 0x08   // джойстик B, ось Y
#define JOY_1_CAL 1      // эта команда калибрует джойстик A
#define JOY_2_CAL 2       // эта команда калибрует джойстик B
unsigned char Buttons(unsigned char button)
(
// эта функция читает статус кнопок джойстика
// сбрасываем содержимое порта 201h
outp (JOYPORT, 0);
// инвертируем прочитанное из порта значение и комбинируем
// его с маской
return (~inp( JOYPORT) & button) ;
}
unsigned char Buttons_Bios (unsigned char button)
{ // чтение кнопок через обращение к BIOS
union _REGS inregs, outregs;
inregs.h.ah == 0x84;    // функция джойстика 84h
inregs.x.dx = 0x00;    // подфункция 0h - чтение кнопок
// вызов BIOS
_int86 (0x15, &inregs, &outr.egs);
// инвертируем полученное значение и комбинируем его с маской
return(~outregs.h.al) & button);
}
```

Теперь посмотрим на детали Листинга 3.1.

- Функция Buttons() и Buttons_Bios() возвращают одинаковый результат. Buttons() посылает 0 в порт джойстика (это делается для того, чтобы инициировать порт) и затем читает данные;
- Как только данные получены, мы маскируем младшие четыре бита и инвертируем четыре старших;
- Этот листинг включает также определение констант (#define), что делает интерфейс более удобным;
- Buttons_Bios() для чтения джойстика использует BIOS. Как только выполняется вызов, результат помещается в регистр AL. В принципе, для таких простых вещей, как кнопки, я использую прямой доступ к портам. Я уже говорил, что использование функций BIOS более медлительно. Правда, по отношению к джойстику это, может быть, и не самый плохой подход. Если вы хотите читать с помощью BIOS - читайте.

Чтение позиции джойстика

Чтение позиции джойстика — весьма утомительная, но вполне выполнимая задача. Все, что нам надо сделать, это послать джойстику простую команду. Это делается записью значения 0 в порт 201h. Затем мы ждем, когда установится нужный нам бит (0-3) порта джойстика. Во время ожидания мы должны включить счетчик. Когда нужный бит установлен, то число, которое мы насчитаем, и есть позиция джойстика. Листинг 3.2 показывает код, который все это делает.

Листинг 3.2. Чтение позиции джойстика.

```
unsigned int Joystick{unsigned char stick)
{
asm {
cli                     ;запретить прерывания
mov ah,byte ptr stick   ;замаскировать AH,
                        ;чтобы выбрать джойстик
xor al,al               ;обнулить AL
xor cx,cx               ;обнулить CX
mov dx,JOYPORT          ;используем DX для ввода и вывода
out dx,al

discharge:
in al,dx                ;читаем данные из порта
test al,ah              ;изменился ли бит запроса?
loopne discharge        ;если нет, повторяем чтение
sti                     ;разрешить прерывания
xor ax, ax              ;обнулить AX
sub ax,cx,              ;теперь AX содержит позицию джойстика
} // конец ассемблерного блока
```

```
// возвращаемое значение содержится в AX
} // конец функции
```

(Кстати, встроенный ассемблер мне все больше и больше нравится.) Программа достаточно проста: при запуске программа обнуляет регистры AX и CX;

- Затем программа опрашивает порт джойстика;
- Далее подсчитывается количество циклов в ожидании, пока установится нужный бит;
- Подсчет выполняется в регистре CX с помощью инструкции LOOPXX (в данном случае используется команда LOOPNE);
- Инструкция TEST определяет установку бита;
- Когда нужный бит установлен, программа выходит из цикла. Результат передается вызывающей программе, в регистре AX.

С этим вроде все. Позже я покажу демонстрационную программу, в которой применяются все функции джойстика. Благодаря обращению к BIOS она обладает лучшей совместимостью, легче переносима и умеет производить самокалибровку. Я думаю, что и вы будете пользоваться ею.

Калибровка джойстика

Теперь разберемся с калибровкой джойстика. Как я уже говорил ранее, результат, который мы получим от чтения джойстика в цикле, будет разным на разных ПК. На одних компьютерах значения по оси X окажется в пределах от 0 до 255, на других - от 0 до 10000.

Таким образом, нам надо нормализовать эти данные или масштабировать их. Стандартным приемом в данных случаях может служить калибровка джойстика самим игроком в setup'е игры. Во время этой процедуры игрок двигает джойстиком, а программа считывает и запоминает данные калибровки где-нибудь на диске для дальнейшего использования.

Для того чтобы проанализировать джойстик, программа должна:

- Найти значения максимального и минимального отклонения по осям X и Y;
- Сохранить эту информацию;
- Использовать полученные данные для выяснения, на какой угол игрок отклонил ручку джойстика.

Например, джойстик был откалиброван и мы обнаружим, что ось X имеет значения от 0 до 255. Затем, если значение джойстика, например, по координате X окажется равным 128, то можно с уверенностью сказать, что рукоятка находится в среднем положении (кстати, в процессе калибровки средняя позиция также запоминается).

Осталась одна маленькая деталь, о которой стоит сказать — это детектирование джойстика (то есть проверка его наличия). Обычная техника детектирования такая: в цикле опрашивается порт джойстика в течение определенного времени. Но надо сказать, что с помощью функций BIOS это можно сделать более надежно. Если после вызова функции значения по всем координатам равны 0, то никакого джойстика нет.

Я думаю, что мы уже сказали о джойстиках все. Теперь напишем небольшую программку, в, которой используем функции из Листингов 3.1 и 3.2. При старте она просит игрока:

- Подвигать джойстиком;
- Установить джойстик в среднее положение;
- Понажимать на кнопки.

Затем, программа сохраняет результаты процедуры калибровки в глобальных переменных.

Листинг 3.3. Программа работы с джойстиком (JOY.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ,////////////////////////////////////
#include <dos.h>
#include <bios.h>
#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <graph.h>
// ОПРЕДЕЛЕНИЯ ////////////////////////////////////////
#define JOYPORT      0x201 // порт джойстика - 201h
#define BUTTON_1_A   0x10 // джойстик A, кнопка 1
#define BUTTON_1_B   0x20 // джойстик A, кнопка 2
#define BUTTON_2_A   0x40 // джойстик B, кнопка 1
#define BUTTON_2_B   0x80 // джойстик B, кнопка 2
#define JOYSTICK_1_X 0x01 // джойстик A, ось X
#define JOYSTICK_1_Y 0x02 // джойстик A, ось Y
#define JOYSTICK_2_X 0x04 // джойстик B, ось X
#define JOYSTICK_2_Y 0x08 // джойстик B, ось Y
```

```
#define JOY_1_CAL       1 // команда калибровки джойстика A
#define JOY_2_CAL       2 // команда калибровки джойстика B
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ ////////////////////////////////////
unsigned int
joy_1_max_x, // глобальные переменные для сохранения
joy_1_max_y, // калибровочных значений
joy_1_min_x,
joy_l_min_y,
joy_1_cx, joy_1_cy,
joy_2_max_x, joy_2_max_y,
joy_2_min_x, joy_2_min_y,
joy_2_cx, joy_2_cy;
// ФУНКЦИИ //////////////////////////////////////////////
unsigned char Buttons(unsigned char button)
{
// функция читает статус кнопок джойстика
outp(JOYPORT,0); // получаем запрос на получение статуса кнопок
// инвертируем полученное значение и комбинируем его с маской
return (~inp(JOYPORT) & button);
} //конец функции ////////////////////////////////////////////////////
unsigned int Joystick(unsigned char stick)
{
// Функция  читает положение счетчика на основании количества
// циклов, прошедших между сбросом и установкой бита в порту
// джойстика. Встроенный ассемблер - прекрасная вещь!
_asm{
cli                     ; запрещаем прерывания
mov ah, byte ptr stick  ; в регистр AH заносим маску
; для выбора джойстика
xor al,al               ; обнуляем AL
xor cx,cx               ; обнуляем CX
mov dx,JOYPORT          ; в DX помещаем номер порта джойстика
out dx,al               ; обнуляем содержимое порта
discharge:
in al,dx                ; читаем данные из порта
test al,ah              ; установился ли бит?
loopne discharge        ; если нет - повторяем чтение
sti                     ; разрешаем прерывания
xor ax,ax               ; обнуляем AX
sub ax,cx               ; теперь AX содержит искомое значение
} // конец ассемблерного блока
} // конец функции
//////////////////////////////////////////////////
unsigned int Joystick_Bios(unsigned char stick)
(
// версия функции чтения состояния джойстика
// работающая через BIOS
union _REGS inregs, outregs;
inregs.h.ah = 0x84; // нам нужна функция 84h
inregs.x.dx = 0x01;
// подфункция 01h - чтение состояния джойстика
_int86(0x15,&inregs, &outregs); // вызываем BIOS
// возвращаем требуемое значение
switch(stick)
{
case JOYSTICK_1_X:
{
return(outregs.x.ax);
} break;
case JOYSTICK1_Y:
{
return(outregs. x. bx} ;
} break;
case JOYSTICK_2_X:
{
return(outregs.x.ex);
} break;
```

```
case JOYSTICK_2_Y:
{
return(outregs.x.dx) ;
} break;
default:break;
} // конец оператора switch
} // конец функции
//////////////////////////////////////////////////
unsigned char Buttons_Bios(unsigned char button)
{
// версия функции для чтения статуса кнопок джойстика,
// работающая через BIOS
union _REGS inregs, outregs;
inregs.h.ah = 0x84; // обращение к джойстику - функция 84h
inregs.x.dx = 0x00; // подфункция 0 - чтение статуса кнопок
_int86 (0x15, &inregs, &outregs);
// инвертируем результат и комбинируем с переданной маской
return((~outregs.h.al) & button);
} // конец функции //////////////////////////////////////////////////
void Joystick_Calibrate(int stick)
{
// функция выполняет калибровку джойстика путем нахождения
// минимального и максимального значений по осям X и У. Затем
// полученные значения сохраняются в глобальных переменных.
unsigned int x_new,y_new; // позиция джойстика
if (stick==JOY_1_CAL) {
printf("\nCalibrating Joystick #1: Swirl stick, then release it and press FIRE");
// придаем переменным заведомо невозможные значения
joy_1_max_x=0;
joy_1_max_y=0;
joy_1_min_x=10000;
joy_1_min_y=10000;
// пользователь должен покрутить джойстик, поставить его в среднее
// положение и затем нажать любую кнопку
while(!Buttons(BUTTON_1_A | BUTTON_1_B))
{
// читаем новые значения потенциометра и пытаемся улучшить
// калибровку
x_new = Joystick_Bios(JOYSTICK_1_X);
y_new = Joystick_Bios(JOYSTICK_1_Y};
// обрабатываем ось X
if (x_new >= joy_1_max x) joy_1_max x = x_new;
if (x_new <= joy_1_min_x) joy_1_min_x = x_new;
// обрабатываем ось Y
if (y_new >= joy_1_max_y) joy_1_max_y = y_new;
if (y_new <= joy_1_ min y) joy_1_min y = y_new;
} // конец цикла while
// получаем значения потенциометра, соответствующие нейтральному
// положению
joy_1_cx = x_new;

joy_l_cy = y_new;
} // конец калибровки джойстика A
else
if (stick==JOY_2_CAL)
{
printf("\nCalibrating Joystick #2: Swirl stick, then release it and pres's FIRE");
// придаем переменным заведомо невозможные значения
joy_2_max x=0;
joy_2_max_y=0;
joy_2_min_x=10000;
joy_2_min_y=10000;
// пользователь должен покрутить джойстик, поставить его в
// нейтральное положение и нажать любую кнопку
while(!Buttons(BUTTON_2_A | BUTTON_2_B))
{
// читаем значение потенциометра и пытаемся улучшить калибровку
```

```
x_new = Joystick (JOYSTICK_2_X) ;
y_new = Joystick(JOYSTICK_2_Y);
// обрабатываем ось X
if (x_new >= joy_2_max_x)
joy_2_max x = x_new;
else if (x_new <= joy_2_min_x)
joy_2_min_x = x_new;
// обрабатываем ось Y
if (y_new >=joy_2_max_y)
joy_2_max_y = y_new;
else if (y_new <= joy_2_min_y)
joy_2_min_y = y_new;
} // конец цикла while
// читаем значения, соответствующие нейтральному положению
joy_2_cx = x_new;
joy_2_cy = y_new;
} // конец калибровки джойстика B
printf ("\nCalibration Complete... hit any key to continue.");
getch();
} // конец функции калибровки джойстика
// ОСНОВНАЯ ПРОГРАММА ////////////////////////////////////
void main(void) (
// калибруем джойстик
Joystick_Calibrate(JOY_1_CAL) ;
_clearscreen(_GCLEARSCREEN);
// даем пользователю поиграть с джойстиком
while(!kbhit())
{
_settextposition(2,0);
printf("Joystick 1 = [%u,%u]    ", Joystick__Bios(JOYSTICK_1_X),
Joystick_Bios(JOYSTICK_1_Y));
if (Buttons_Bios(BUTTON_1_A))
printf("\nButton 1 pressed   ");
else if (Buttons_Bios(BUTTON_1_B))
printf("\nButton 2 pressed   ");
else
printf("\nNo Button Pressed ") ;
} // конец цикла while
// даем пользователю посмотреть, на калибровочные данные
printf("\nmax x=%u/ max y=%u,min x=%u,min y=%u, cx=%u, cy=%u", joy_1_max_x,  joy_1_max_y,
joy_1_min_x, joy_1_min_y, joy_1_cx, joy_1_cy) ;
// кое-что будем добавлять позже
} // конец функции main
```

Если вы введете программу с Листинга 3.3, то увидите, как джойстик A изменяет свои значения в процессе работы с ним.

Клавиатура

Клавиатура - это наиболее сложное устройство ввода, которое есть в ПК. Она даже имеет свою собственную микросхему - контроллер ввода. Я провел много бессонных ночей, вчитываясь в листинги BIOS и пытаясь понять тайны, скрытые в работе с клавиатурой.

В этой жизни есть множество непонятных вещей - курс доллара, термический - коэффициент расширения рубидия и т. д. Несомненно одно - любовь людей к клавиатуре абсолютно необъяснима.

Для наших целей (для написания видеоигр) мы должны научиться хорошо работать с клавиатурой. Для этого вовсе не стоит разбираться с прерываниями, регистрами и портами. Мы будем использовать функции языка Си и BIOS для работы с очередью клавиатуры. Говоря о Си, я не имею в виду функции типа getch () и scanf (). Речь пойдет, скорее, о функциях типа _bios_keyboard ().

Примечание

Давайте приостановимся и немного подумаем, Общее правило для авторов игр - никогда не использовать BIOS. Верно? Хорошо, на самом деле BIOS вполне можно использовать для файловых операций и для выделения памяти. В общем, обращения к BIOS вполне допустимы в функциях, некритичных по времени. Попытка использовать его для работы с джойстиком или клавиатурой не будет для нас смертельна (в отличие от попыток организовать через BIOS вывод графики). Как я уже говорил, современные компьютеры достаточно быстры, чтобы нам не приходилось

оптимизировать каждую запятую в тексте программы или писать ее целиком на ассемблере.

BIOS поддерживает несколько функций, которые мы будем использовать и которые приведены в таблице 3-1.

Таблица 3.1. Клавиатурные функции BIOS.

Bios INT 16h

Функция 00h - чтение символа с клавиатуры.

Вход: AH: 00h

Выход: AH - скан код

AL - ASCII-символ

Функция 01h - чтение статуса клавиатуры.

Вход: AH: 01h

Выход: AH - скан-код

AL - ASCII-символ

флаг **Z:** если 0, то в буфере есть символ, если 1 - нет символа.

Функция 02h - Флаги, возвращаемые клавиатурой.

Вход: AH: 02h

Выход: AL - байт статуса клавиатуры:

бит 0 - нажат правый **Shift;**

бит 1 - нажат левый **Shift;**

бит 2 - нажата клавиша **Ctrl;**

бит 3 - нажата клавиша **Alt;**

бит 4 - Scroll Lock в положении **ON;**

бит 5 - Num Lock в положении **ON;**

бит 6 - Caps Lock в положении **ON**;

бит 7 - Insert в положении **ON.**

Скан-коды

Давайте теперь поговорим о такой вещи как скан-коды. Если вы считаете, что при нажатии клавиши A обработчик клавиатуры также получает код символа A, то вы ошибаетесь. К сожалению, это не так. Обработчику посылается скан-код. Более того, он посылается дважды — при нажатии и отпускании клавиши. В видеоиграх нас будут интересовать не столько ASCII-коды, сколько нажатия клавиш A, S, Пробел, которые обычно отвечают за маневры, стрельбу и т. д. Таким образом, нам надо знать, как получить именно скан-коды. И это все, что требуется. В таблице 3.2 перечислены скан-коды клавиш.

Таблица 3.2. Таблица скан-кодов.

Клавиша	Скан-код	Клавиша	Скан-код	Клавиша	Скан-код	Клавиша	Скан-код
Esc	1	**I**	22	**Z**	43	**F7**	64
1	2	**O**	23	**X**	44	**F8**	65
2	3	**P**	24	**C**	45	**F9**	66
4	4	**[**	25	**V**	46	**F10**	67
5	5	**]**	26	**B**	47	**F11**	133
6	6	**Enter**	27	**N**	48	**F12**	134
7	7	**Ctrl**	28	**M**	49	**Num Lock**	69
8	8	**A**	29	**Запятая**	50	**Scroll Lock**	70
9	9	**S**	30	**Точка**	51	**Home**	71
0	10	**D**	31	**/**	52	**Up**	72
-	11	**F**	32	**Правый Shift**	53	**PgUp**	73
=	12	**G**	33	**Print Screen**	54	**Серый -**	74

Backspace	13	**H**	34	**Alt**	55	**Left**	75
Tab	14	**J**	35	**Пробел**	56	**5 на цифр. клав.**	76
Q	15	**K**	36	**Caps Lock**	57	**Right**	77
W	16	**L**	37	**F1**	58	**Серый +**	78
E	17	**;**	38	**F2**	59	**End**	79
R	18	**Апостроф**	39	**F3**	60	**Down**	80
T	19	**~**	40	**F4**	61	**PgDn**	81
Y	20	**Левый Shift**	41	**F5**	62	**Ins**	82
U	21	****	42	**F6**	63	**Del**	83

Если вы внимательно изучали таблицу, то должны, были заметить, что клавиши; имеющие двухсимвольную кодировку, обладают, тем не менее, только одним скан-кодом. Это происходит потому, что каждый скан-код может быть дополнен информацией о статусе клавиш. Кроме того, благодаря таблице 3.2, мы теперь сами можем по скан-коду определять код ASCII.

Статус клавиш

Мы должны иметь возможность определять:

- Была ли нажата какая-нибудь клавиша;
- Какая была нажата клавиша;
- Статус клавиши Shift.

Статус клавиш — это просто битовый вектор (последовательность), содержащий информацию о клавишах Shift, Alt, Ctrl и других. Эта последовательность находится в памяти по адресам 417h и 418h. Мы не будем читать эти ячейки напрямую, а воспользуемся BIOS и Си.

Листинг 3.4 содержит код, позволяющий получить статус клавиш.

Листинг 3.4. Получение статуса клавиш.

```
#define SHIFT_R         0x0001
#define SHIFT_L         0x0002
#define CTRL            0x0004
#define ALT             0x0008
#define SCROLL_LOCK_ON  0x0010
#define NUM_LOCK_ON     0x0020
#define CAPS_LOCK_ON    0x0040
#define INSERT_MODE     0x0080
#define CTRL_L          0x0100
#define ALT_L           0x0200
#define CTRL_R          0x0400
#define ALT_R           0x0800
#define SCROLL_LOCK_DWN 0x1000
#define NUM_LOCK_DWN    0x2000
#define CAPS_LOCK_DWN   0x4000
#define SYS_REQ_DWN     0x8000
unsigned int Get_Control_Keys(unsigned int mask)
{
// функция возвращает статус интересующей нас управляющей клавиши
return(mask &_bios_keybrd(_KEYBRD_SHIFTSTATUS));
} // конец функции
```

В листинге 3.4 функция Get_Control_Key() использует вызов BIOS из Си для определения статуса клавиш. В строки #define включены описания масок для определения статусных клавиш, благодаря чему вы можете вызывать функцию Get_Control_Key(), не задумываясь о значении битов состояния. Более того, используя маски и логический оператор AND, за один вызов можно получить сразу несколько состояний.

Получение скан-кодов с клавиатуры

Код, представленный в Листинге 3.5 напрямую считывает скан-код и возвращает его в вызывающую программу. Если ввода нет, то функция возвращает 0.

Листинг 3.5. Получение скан-кодов с клавиатуры.

```
unsigned char Get_Scan_Code(void)
{
// получить скан-код нажатой клавиши
// используется встроенный ассемблер
```

```
//клавиша нажата?
_asm
{
mov ah,01h    ;функция Olh - проверка на нажатие клавиш
int 16h       ;вызвать прерывание
jz empty      ;нет нажатых клавиш – выходим
mov ah,00h    ;функция 0 - получить скан-код
int 16h       ;вызвать прерывание
mov al,ah     ;результат поместить в AL
xor ah,ah     ;обнуляем AH
jmp done      ;в AX возвращается значение "все в порядке"
empty:
xor ax,ax     ;очистить AX
done:
} // конец ассемблерного блока
} // конец функции
```

Мы опять используем встроенный ассемблер. Можно, было, конечно, использовать функцию BIOS через вызов _int86() в Си, но на встроенном ассемблере это выглядит намного круче.

Получение ASCII-кодов с клавиатуры

Давайте теперь посмотрим, как мы можем получить ASCII-символ, введенный с клавиатуры. Это может быть полезно, когда игрок вводит свое имя и нам нужны ASCII-коды. Мы можем получить скан-коды и транслировать их в ASCII, но к чему такие сложности, если сразу можно прочитать ASCII-коды?

Листинг 3.6 показывает функцию, которую мы будем часто использовать, работая с клавиатурой. Эта программа опрашивает клавиши и определяет их нажатие. Если символ введен, то функция возвращает его ASCII-код, в противном случае возвращается 0.

Листинг 3.6. Получение ASCII-кодов с клавиатуры.

```
unsigned char Get_Ascii_Key(void)
{
//если это нормальный ascii код – возвращаем его, иначе 0
if (_bios_keybrd(_KEYBRD_READY))
      return(_bios_keybrd(_KEYBRD_READ));
else
      return(0);
}// конец функции
```

Чтобы использовать функцию из Листинга 3.6, вы должны выполнить примерно следующие действия:

```
if (( c=Get_Ascii_Key()) > 0)
{
      обработать_символ
} иначе
{
      символов_нет
}
```

Теперь все вместе: Демонстрационная программа работы с клавиатурой

Как теперь насчет того, чтобы собрать все написанное про клавиатуру в одну кучу? В Листинге 3.7 представлена демонстрационная программа, которая состоит из вызовов уже написанных в этой главе функций. Она показывает скан-коды нажатых клавиш и состояние клавиш Ctrl и Alt. Если вы нажмете Q, программа завершит свою работу.

Листинг 3.7. Демонстрационная программа работы с клавиатурой (KEY.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ///////////////////////////////////////
#include <dos.h>
#include <bios.h>
#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <graph.h>
// ОПРЕДЕЛЕНИЯ /////////////////////////////////////////////
// битовые маски для управляющих клавиш
#define SHIFT_R             0x0001
#define SHIFT_L             0x0002
```

```
#define CTRL                0x0004
#define ALT                 0x0008
#define SCROLL_LOCK_ON      0x0010
#define NUM_LOCK_ON         0x0020
#define CAPS_LOCK_ON        0x0040
#define INSERT_MODE         0x0080
#define CTRL_L              0x0100
#define ALT_L               0x0200
#define CTRL_R              0x0400
#define ALT_R               0x0800
#define SCROLL_LOCK_DWN     0x1000
#define NUM_LOCK_DWN        0x2000
#define CAPS_LOCK_DWN       0x4000
#define SYS_REQ_DWN         0x8000
// Значения скан-кодов. Внимание: каждая клавиша продуцирует только
// один скан-код, поэтому определения даны только для символов
// нижнего регистра. Например, одна и та же клавиша соответствует
// символу "1" (нижний регистр) и символу "!" (верхний регистр).
// Однако пользоваться надо все равно определением SCAN_1
#define SCAN_ESC          1
#define SCAN_1            2
#define SCAN_2            3
#define SCAN_3            4
#define SCAN_4            5
#define SCAN_5            6
#define SCAN_6            7
#define SCAN_7            8
#define SCAN_8            9
#define SCAN_9            10
#define SCAN_0            11
#define SCAN_MINUS        12
#define SCAN_EQUALS       13
#define SCAN_BKSP         14
#define SCAN_TAB          15
#define SCAN_Q            16
#define SCAN_W            17
#define SCAN_E            18
#define SCAN_R            19
#define SCAN_T            20
#define SCAN_Y         21
#define SCAN_U         22
#define SCAN_I         23
#define SCAN_0         24
#define SCAN_P         25
#define SCAN_LFT_BRACKET 26
#define SCAN_RGT_BRACKET 27
#define SCAN_ENTER     28
#define SCAN_CTRL      29
#define SCAN_A         30
#define SCAN_S         31
#define SCAN_D         32
#define SCAN_F         33
#define SCAN_G         34
#define SCAN_H         35
#define SCAN_J         36
#define SCAN_K         37
#defane SCAN_L         38
#define SCAN_SEMI      39
#define SCAN_APOS      40
#define SCANJTILDE     41
#define SCAN_LEFT_SHIFT 42
#define SCAN_BACK_SLASH 43
#define SCAN_Z         44
#define SCAN_X         45
#define SCAN_C         46
#define SCAN_V         47
#define SCAN_B         48
```

```
#define SCAN_N         49
#define SCAN_M         50
#define SCAN_COMMA     51
#define SCAN_PERIOD    52
#define SCAN_FOWARD_SLASH 53
#define SCAN_RIGHT_SHIFT 54
#define SCAN_PRT_SCRN  55
#define SCAN_ALT       56
#define SCAN_SPACE     57
#define SCAN_CAPS_LOCK 58
#define SCAN_F1        59
#define SCAN_F2        60
#define SCAN_F3        61
#define SCAN_F4        62
#define SCAN_F5        63
#define SCAN_F6        64
#define SCAN_F7        65
#define SCAN_F8        66
#define SCAN_F9        67
#define SCAN_F10       68
#define SCAN_Fll       133
#define SCAN_Fl2       134
#define SCAN_NUM_LOCK  69
#define SCAN_SCROLL_LOCK  70
#define SCAN_HOME         71
#define SCAN_UP           72
#define SCAN_PGUP         73
#define SCAN_NUM_MINUS    74
#define SCAN_LEFT         75
#define SCAN_CENTER       76
#define SCAN_RIGHT        77
#define SCAN_NUM_PLUS     78
#define SCAN_END          79
#define SCAN_DOWN         80
#define SCAN_PGDWN        81
#define SCAN_INS          82
#define SCAN_DEL          83
// ФУНКЦИИ ////////////////////////////////////////////
unsigned char Get_Ascii_Key(void)
{
// Если в буфере клавиатуры есть символ, функция возвращает его
// ASCII-код. Если символа нет, возвращается 0.
if (_bios_keybrd(_KEYBRD_READY)) return(_bios_keybrd(_KEYBRD_READ)) ;
else return(0);
} // конец функции
//////////////////////////////////////////////////////
unsigned int Get_Control_Keys(unsigned int mask)
(
// функция возвращает состояние любой управляющей клавиши
return(mask & _bios_keybrd(_KEYBRD_SHIFTSTATUS));
}//конец функции ////////////////////////////////////////////////////
unsigned char Get_Scan_Code(void) (
// функция возвращает скан-код нажатой клавиши // используется встроенный ассемблер
_asm{
mov ah,01h    ;функция 01h - проверка нажатия клавиши
int 16h       ;вызов прерывания
jz empty      ;нет символа - выход
mov ah,00h    ;функция 0 - получение скан-кода
int 16h       ;вызов прерывания
mov al,ah     ;перемещаем результат из AH в AL
xor ah,ah     ;обнуляем AH
jmp done      ;результат возвращается в AX
empty:
xor ax,ax    ;обнуляем AX
done:
} //конец ассемблерного блока
} // конец функции
```

```
// ОСНОВНАЯ ПРОГРАММА ////////////////////////////////////
void main(void)
{
unsigned char key;
int done=0;
unsigned int control;
_clearscreen(_GCLEARSCREEN);
while(!done)
{
_settextposition(2,0);
if ( (key = Get_Scan_Code()) ) printf("%c %d  ",key,key);
// проверка на нажатие Ctrl и Alt
if (Get_Control_Keys(CTRL))
printf("\ncontrol key pressed");
if (Get_Cbntrol_Keys(ALT))
printf("\nalt key pressed    ");
if (key==16) done=1); // 16 – это скан-код клавиши Q
}// конец цикла while
} // конец функции main
```

Мышь

Изобретение мыши, безусловно, было событием в компьютерной истории. Появившись в результате исследований в центре Xerox PARK в Калифорнии, она завоевала сердца миллионов пользователей. Мышь позволила наглядно работать с экраном и легко общаться с интересующими объектами. Обычно мышь подсоединяется к последовательному порту компьютера или DIN коннектору на лаптопе. Существующая *bus mice* требует специальной платы, поэтому не получила широкого распространения (это понятно — мало кто хочет просто так вскрывать свой компьютер для установки одной платы).

На сегодняшний день существуют как механические, так и оптические мыши. В обоих случаях перемещение мыши кодируется в последовательность импульсов, которая преобразуется в пакет, содержащий информацию о движении мыши и состоянии кнопок. Этот пакет посылается в серийный порт, к которому подсоединяется мышь, а затем интерпретируется программой.

Вообще-то, написание драйвера мыши — не самое скучное занятие, но мы этого делать не будем. К счастью, Microsoft и другие фирмы написали кучу подобных драйверов. Поэтому мы не станем самостоятельно декодировать мышиные пакеты, а воспользуемся более удобным программным интерфейсом.

Мы будем использовать только минимум функций для определения позиции мыши и статуса кнопок. В таблице 3.3 перечислены эти функции.

Замечание

Микки (mickey) — это самое маленькое расстояние, которое отслеживается мышью. Оно примерно равно 1/200 дюйма.

Таблица 3.3. Функции драйвера мыши.

Bios INT 33h

Функция 00h - инициализировать драйвер мыши
Вход: AX: 0000h
Выход: AX: FFFFh в случае успеха,
0000h при неудаче
BX - количество кнопок мыши

Функция 01h - включить курсор мыши
Вход: AX: 0001h
Выход: Ничего

Функция 02h - выключить курсор мыши
Вход: AX: 0002h
Выход: Ничего

Функция 03h - возвратить позицию курсора и статус клавиш
Вход: AX:0003h
Выход: BX - статус кнопок
Бит 0 - левая кнопка: 1 - нажата, 0 - не нажата
Бит 1 - правая кнопка: 1 - нажата, 0 - не нажата
Бит 2 - центральная кнопка: 1 - нажата, 0 - не нажата
CX - X-координата курсора
DX - Y-координата курсора

Функция 0Bh - возвратить относительную позицию мыши

Вход: AX: 000Bh
Выход: CX - относительное горизонтальное движение в mickey
DX - относительное вертикальное движение в mickey

Функция 1Ah - установить чувствительность

Вход: AX: 00lAh
Выход: BX - чувствительность по оси X (0-100)
CX - чувствительность по оси Y (0-100)
DX - значение скорости, при которой чувствительность возрастает в 2 раза (0-100)

Как видите, функции драйвера вызываются через прерывание 33h. Мы записываем параметр в регистр AX .и получаем результат в регистрах AX, BX, CX и DX. Я написал простую функцию для работы с мышью, она называется Squeeze_Mouse(). Эта функция может выполнять много действий — все зависит от передаваемых параметров. Прототип функции:

```
int Squeeze_Mouse(int command, int *x, int *y, int *buttons) ;
```

Кроме этого, я сделал несколько описаний, чтобы упростить работу с ней:

```
#define MOUSE_INT             0x33  // номер прерывания
#define MOUSE_RESET           0x00  // сброс мыши
#define MOUSE_SHOW            0x01  // показать мышь
#define MOUSE_HIDE            0x02  // погасить мышь
#define MOUSE_BUTT_POS        0x03  // возвратить координаты
                                    //и количество кнопок
#define MOUSE_SET_SENSITIVITY 0x1A  // установить
                                    //чувствительность
                                    //в пределах 0-100
#define MOUSE_MOTION_REL      0x0B  // установить
                                    // относительную
                                    // чувствительность
```

Таким образом, если мы хотим получить координаты мыши, то должны написать следующее:

```
Squeeze Mouse(MOUSE_BUTT_POS, &mouse_x, &mouse_y, &mouse_buttons);
```

где mouse_x, mouse_y и mouse_buttons - локальные переменные для сохранения результатов.

Теперь обратим внимание на два способа, используемые для передачи координат мыши:

- Драйвер мыши может возвращать абсолютные координаты. В этом случае значения X и Y являются координатами мыши на экране. К примеру, если мышь находится в левом верхнем углу экрана, функция возвращает (0,0);
- Драйвер мыши может возвращать относительные координаты. При этом возвращается разница координат от предыдущей посылки. Например, если мышь подвинулась на 20 микки по оси X и на 10 по оси Y, то эти значения и будут возвращены. Для чтения в относительном режиме используйте константу MOUSE_MOTION_REL.

Еще несколько слов о мыши. Вы можете менять ее чувствительность к передвижению, используя константу MOUSE_SET_SENSITIVITY. Для этого подберите для переменных X и Y значение от 1 до 100 и вызовите Squeeze_Mouse. Чувствительность мыши определяется как отношение пиксельного перемещения курсора мыши к одному микки.

Листинг 3.8 содержит демонстрационную программу, которая показывает использование мыши. Эта программа позволяет рисовать на экране, нажимая на левую кнопку мыши и менять цвет, используя правую кнопку.

Листинг 3.8. Работа с мышью (MOUSE.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ///////////////////////////////////////
#include <dos.h>
#include <bios.h>
#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <graph.h>
// ОПРЕДЕЛЕНИЯ ////////////////////////////////////////////
// вызовы функций мыши
#define MOUSE_INT             0x33 // Номер прерывания мыши
#define MOUSE_RESET           0x00 // Инициализация драйвера
#define MOUSE_SHOW            0x01 // Показать курсор мыши
#define MOUSE_HIDE            0x02 // Спрятать курсор мыши
#define MOUSE_BUTT_POS        0x03 // Получение полного статуса
#define MOUSE_SET_SENSITIVITY 0x1A // Установка чувствительности
#define MOUSE_MOTION_REL      0x0B // Получить значение счетчика
```

```
                                    // микки
#define MOUSE_LEFT_BUTTON     0x01 // левая кнопка
#define MOUSE_RIGHT_BUTTON    0x02 // правая кнопка
#define MOUSE_CENTER_BUTTON   0x04 // средняя кнопка
// ФУНКЦИИ ////////////////////////////////////////////
int Squeeze_Mouse(int command, int *x, int *y,int *buttons)
{
// Мы будем использовать _int86 вместо встроенного ассемблера
// Почему? И сам не знаю
union _REGS inregs, outregs;
switch(command) {
case MOUSE_RESET:
{
inregs.x.ax = 0x00; // подфункция 0 – инициализация
_int86(MOUSE_INT, &inregs, &outregs};
*buttons = outregs.x.bx; // возвратить количество кнопок
return(outregs.x.ax);    // возвратить общий результат
} break;
case MOUSE_SHOW:
     {
// эта функция инкрементирует счетчик драйвера. Когда значение
// счетчика становится больше или равно 0, курсор появляется
// на экране
inregs.x.ax = 0x01; // подфункция 1 - показать курсор
_int86(MOUSE_INT, &inregs, &outregs);
return (1);
} break;
case MOUSE_HIDE:
{
// эта функция декрементирует счетчик драйвера; когда его
// значение становится меньше 0, курсор исчезает с экрана
inregs.x.ax = 0x02; // подфункция 2 - спрятать курсор
_int86(MOUSE_INT, &inregs, &outregs);
return(1);
} break;
case MOUSE_BUTT_POS:
{
// эта функция  позволяет получить полный статус состояния мыши,
// включая абсолютную позицию курсора в координатах (x,y) и
// состояние кнопок
inregs.x.ax = 0x03; // подфункция 3 - получить статус мыши
_int86(MOUSE_INT, &inregs, &outregs);
// извлечь информацию и вернуть ее через указатели
*x       = outregs.x.cx;
*y       = outregs.x.dx;
*buttons = outregs.x.bx;
return(1);
} break;
case MOUSE_MOTION_REL:
{
// эта функция позволяет получить относительное изменение
// координат мыши с момента последнего вызова
inregs.x.ax = 0x03; // подфункция 1 – получить
                    // относительную позицию
_int86(MOUSE_INT, &inregs, &outregs);
// результат при помощи указателей помещается в переменные x и y
*x       = outregs.x.cx;
*y       = outregs.x.dx;
return(1);
} break;
case MOUSE_SET_SENSITIVITY:
{
// эта функция устанавливает чувствительность мыши. Перед
// вызовом необходимо установить переменные x и y в значения
// из диапазона 1-100. Переменная "buttons" используется для
// установки значения порога удвоения скорости
// (из диапазона 1-100)
```

```
inregs.x.bx = *x;
inregs.x.cx = *y;
inregs.x.dx = *buttons;
inregs.x.ax = 0x1A; // подфункция 26 - установка
                    //  чувствительности
_int86(MOUSE INT, &inregs, &outregs);
return(1);
} break;
default:break;
} // конец оператора switch
} // конец функции
// ОСНОВНАЯ ПРОГРАММА
////////////////////////////////////////////////////////////////////////////
void main(void)
{
int x,y,buttons,num_buttons;
int color=l;
_setvideomode(_VRES16COLOR};  // 640x480, 16 цветов
// инициализация драйвера мыши
Squeeze_Mouse(MOUSE_RESET,NULL,NULL,&num_buttons);
//показать курсор
Squeeze_Mouse(MOUSE_SHOW,NULL,NULL,NULL);
while(!kbhit())
{
_settextposition(2,0);
Squeeze_Mouse(MOUSE_BUTT_POS,&x,&y,&buttons);
printf("mouse x=%d y=%d buttons=%d    ",x,y,buttons) ;
// рисование
if (buttons==1)
{
_setcolor(color) ;
_setpixel(x-1,y-2);
_setpixel(x,y-2) ;
_setpixel(x-1,y-1);
_setpixel(x,y-1);
} // конец обработки нажатия левой кнопки
// выбор цвета
if (buttons==2)
{
if (++color>15) color=0;
// ждем отпускания правой кнопки
while(buttons==2)
{
Squeeze_Mouse(MOUSE_BUTT_POS,&x,&y,&buttons) ;
} // конец ожидания
} // конец графической работы
} // конец цикла while
// назад в текстовый режим
_setvideomode(_DEFAULTMODE);
} // конец функции main
```

ИТОГ

В этой главе мы изучили разные устройства ввода, которые могут использоваться в видеоиграх: клавиатура, джойстик, мышь. Более того, мы даже написали функции для работы с этими устройствами. Еще мы выяснили, что не стоит использовать BIOS для работы с устройствами ввода, за исключением клавиатуры.

МЕХАНИЗМЫ ДВУХМЕРНОЙ ГРАФИКИ

Вы можете считать, что экран компьютера подобен листу бумаги, который можно изгибать, сворачивать в трубочку и вообще как угодно трансформировать в руках. Чтобы делать подобные вещи, мы должны знать язык, с помощью которого все это выполняется — язык математики.

В данной главе мы познакомимся с тем, что называется *двухмерная проекция* и изучим следующие темы:

- Картезианская, или двухмерная проекция;
- Точки, линии и области;
- Трансляция;
- Масштабирование;
- Повороты;
- Разрезание;

Использование матриц. Двухмерная проекция

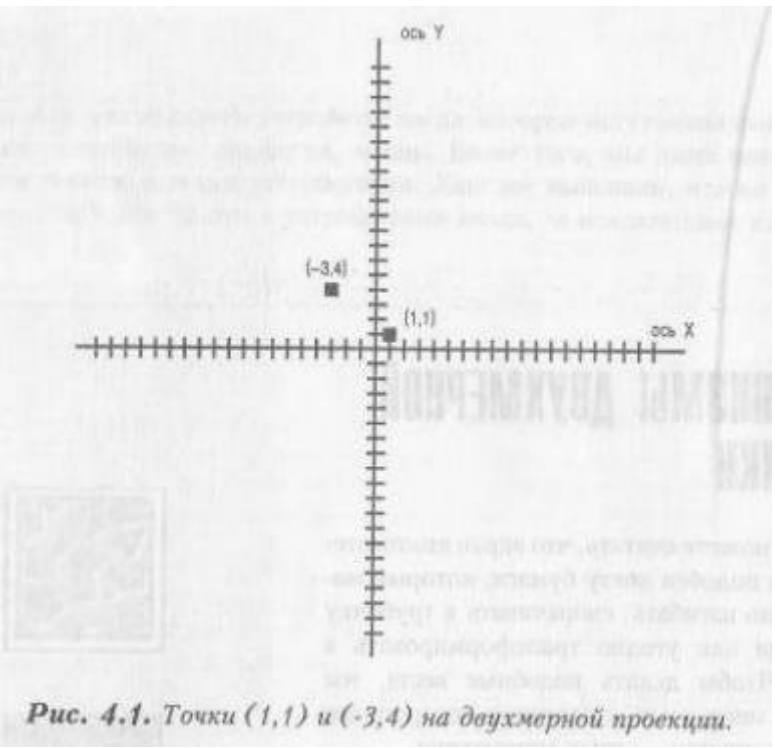


Рис. 4.1. *Точки (1,1) и (-3,4) на двухмерной проекции.*

Двухмерную проекцию (2-D), иначе называемую *картезианской,* можно сравнить с листом бумаги в клеточку.

Каждая точка в двухмерной проекции однозначно описывается двумя координатами. Обычно эти координаты обозначаются буквами x и y, где x определяет точку на горизонтальной оси X, а y задает точку на вертикальной оси Y. Например, если нам захочется нарисовать точки (1,1) и (-3,4), то мы сделаем так, как это показано на рисунке 4.1.

Начав с изучения свойств двухмерной проекции, мы будем постепенно двигаться от простого к сложному, и к тому времени, когда дойдем до трехмерных объектов, они не покажутся чересчур сложными. Кроме того, существует ведь немало весьма привлекательных двухмерных игр, о которых также не стоит забывать. Тем важнее понять механизмы получения двухмерной графики и рассмотреть алгоритмы различных графических преобразований.

Точки, линии и области

Все мы видели игры типа Asteroids, Spectre и Major Havoc. Многие из них имеют общие черты первых видеоигр — все они выполнены линиями и все они, как правило, плоские. Кстати, в предыдущих главах мы еще ничего не делали для рисования проекции кроме отображения точки.

Точки

Мы уже дали определение точке. Она представляет собой позицию на плоскости, которую можно описать парой координат X и Y. Давайте напишем маленькую программу на Си, рисующую точки на экране. Листинг 4.1 показывает такую программу.

Листинг 4.1. Программа, рисующая точки (POINTY.C).

```
#include <stdio.h>
#include <graph.h>
void main(void)
{
int x, y, index, color;
// перевести компьютер в графический режим
_setvideomode(_VRES16COLOR); // режим 640x480, 16 цветов
// нарисовать 10000 точек на экране, расположенных случайным образом
```

```
for(index = 0; index<10000; index++)
{
// получить случайные координаты и цвет
x = rand()%640;
y = rand()%480;
color = rand()%16;
_setcolor(color); // установить цвет для рисования точки
_setpixel(x,y);    // нарисовать точку
} // конец цикла for
// ждать нажатия клавиши
while(!kbhit()){}
// восстановить текстовый режим
_setvideomode(_DEFAULTMODE) ;
} // конец функции main
```

Теперь разберемся, что делает эта программа:

Компьютер переводится в режим VGA с помощью вызова функции Си _setvideomode (VRES16COLOR). Это функция из графической библиотеки Microsoft. После этого программа входит в главный цикл. В структуре

FOR каждый раз случайным образом генерируются три числа: одно для цвета и два других для координат (x,y) позиции точки, которую мы хотим нарисовать:

- Затем мы используем библиотечную функцию _setpixel(x,y), чтобы нарисовать точку на экране. Программа делает это 10000 раз, а потом останавливается;
- Затем программа ждет нажатия любой клавиши, после чего происходит выход в DOS.

Если вы запустите программу несколько раз, то сможете заметить, что точки все время оказываются в одних и тех же местах. Как это получается? Дело в том, что мы пользуемся функцией rand (), которая не является в полном смысле генератором случайных чисел. Она возвращает так называемые псевдослучайные числа. Чтобы избежать этого, вам надо всякий раз при запуске *устанавливать* генератор случайных чисел с разными начальными значениями. Вставьте в начало программы функцию srand(int) — и все будет в порядке,

Линии

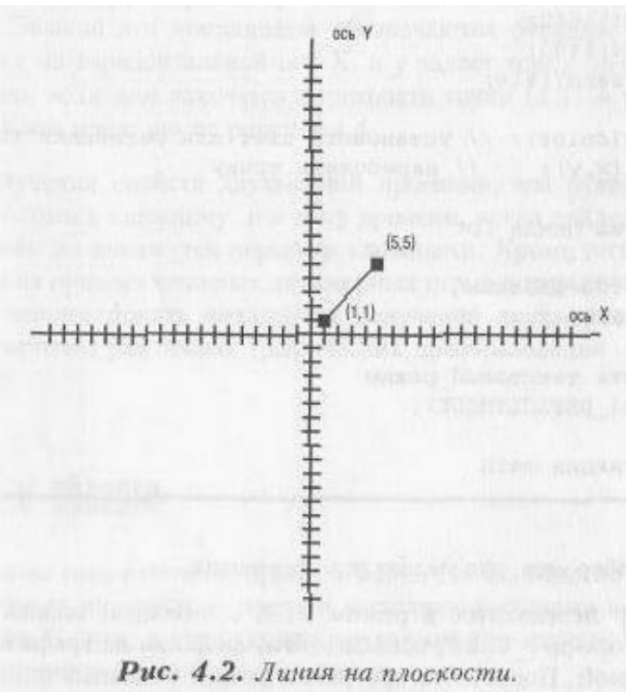


Рис. 4.2. *Линия на плоскости.*

Линия, как вы знаете, — это кратчайший отрезок между двумя точками. Например, между точками (1,1) и (5,5) на плоскости линия будет выглядеть так (рис. 4.2):

Давайте изменим программу из Листинга 4.1 так, чтобы она рисовала линии вместо точек. Чтобы сделать это, нужно кое-что изменить. Вместо двух случайных чисел x и y теперь их будет четыре: (x1,y1) и (x2,y2). Потом программа нарисует между ними линию, используя вызов библиотечной функции. Листинг будет выглядеть так:

Листинг 4.2. Программа, рисующая линии (LINER.C).

```
#include <stdio.h>
#include <graph.h>
void main(void)
{
```

```
int xl,yl,x2,y2,color,index;
// перевести компьютер в графический режим
_setvideomode(_VRES16COLOR); // режим 640x480, 16 цветов
// нарисуем 1000 случайных линий на экране
for (index = 0; index<1000; index++)
{
// получим случайные координаты концов линий и цвет
x1 = rand()%640; // X-координата начальной точки
y1 = rand()%480; // Y-координата начальной точки
x2 = rand()%640; // X-координата конечной точки
y2 = rand()%480; // У-координата конечной точки
color = rand()%16;
_setcolor(color);    // установить цвет
_moveto(x1,y1);    // переместиться к началу линии
_lineto(x2,y2);      // нарисовать линию
} // конец цикла for // ждать нажатия любой клавиши
while(!kbhit()){}
// перевести компьютер в текстовый режим
_setvideomode(_DEFAULTMODE);
} // конец функции main
```

Многоугольники

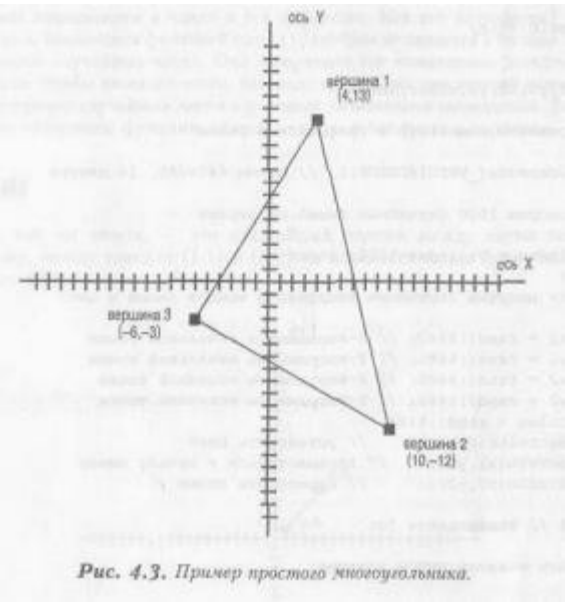


Рис. 4.3. Пример простого многоугольника.

Линии весьма просты, и если вы приложите немного усилий, то сможете из программы 4.2 сделать простой Screen Saver. Но видеоигры кроме линий содержат еще множество интересных графических объектов, например, многоугольников.

Многоугольник — это множество точек, объединенных линиями. Точки пересечения линий называются вершинами многоугольника. На рисунке 4.3 показан треугольник, образованный тремя вершинами.

Все многоугольники являются закрытыми геометрическими объектами. Многоугольники бывают двух типов. Оба типа представлены на рисунке 4.4.

Существует математический алгоритм, позволяющий определить, к какому типу относится данный многоугольник, но он довольно сложен и сейчас нам не нужен. Давайте лучше напишем программу, рисующую многоугольники (Листинг 4.3).

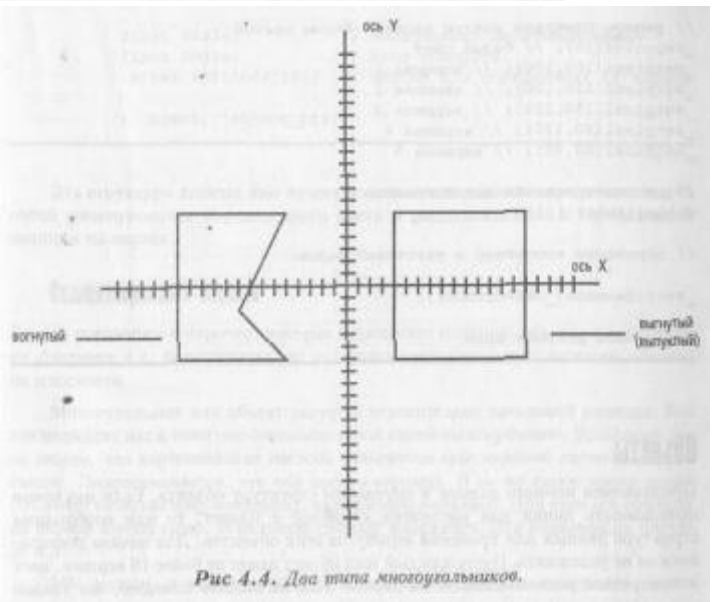


Рис 4.4. Два типа многоугольников.

Листинг 4.3. Программа, рисующая многоугольники (POLYDRAW.C).

```
#include <stdio.h>
#include <graph.h>
void main(void)
{
// перевести компьютер в графический режим
_setvideomode( VRES16COLOR); // режим 640x480, 16 цветов
// нарисовать простой многоугольник
_setcolor(1);        // пусть он будет синего цвета
_moveto(100,100); // первая вершина
_lineto(120,120);   // вторая вершина
_lineto(150,200);   // третья вершина
_lineto(SO,190);    // четвертая вершина
_lineto(90,60);     // пятая вершина
_lineto (100,100); // назад для замыкания контура
// теперь отмечаем каждую вершину белым цветом
_setcolor(15); // белый цвет
_setpixel(100,100); // вершина 1
_setpixel(120,120); // еершина 2
_setpixel(150,200); // вершина 3
_setpixel(80,190); // вершина 4
_setpixel(80,60); // вершина 5
// ожидание нажатия любой клавиши
while(!kbhit()){}
// перевести компьютер в текстовый режим
_setvideomode(_DEFAULTMODE);
} // конец функции main
```

Объекты

Продвинемся немного дальше и определим структуру объекта. Если мы хотим использовать линии для рисования кораблей и планет, то нам необходима структура данных для хранения атрибутов этих объектов. Для начала постараемся ее не усложнять. Пусть каждый наш объект имеет не более 16 вершин, цвет и конкретное местоположение на экране. Как вычислять позицию, мы узнаем позже, а код для описания вершин в структуре данных будет выглядеть примерно так:

Листинг 4.4. Структуры данных для задания вершин объектов.

```
typedef struct vertex_typ
{
float x,y;
} vertex, *vertex_ptr;
// структура объекта
typedef struct object_typ
{
int num_vertices;     // количество вершин в объекте
int color;            // цвет объекта
float xo,yo;          // позиция объекта
float x_velocity;     // используем позже для
float y_velocity;     // перемещения объекта
float scale;          // коэффициент масштабирования
float angle;          // угол поворота
vertex vertices[16];  // массив для определения 16 вершин
} object, *object_ptr;
```

Эта структура данных нам нужна для описания объекта, представляющего собой многоугольник определенного цвета и расположенного в определенной позиции на экране.

Позиционирование объекта

Теперь поговорим о строчке, которая определяет позицию объекта в структуре из Листинге 4.4. Координаты (xo,yo) описывают начальную позицию объекта на плоскости.

Многоугольник или объект рисуется относительно начальной позиции. Все это подводит нас к понятию относительной системы координат. Возможно, вы не знаете, что картезианская система называется еще мировой системой координат. Подразумевается, что она просто огромна. В то же самое время экран ПК имеет свою систему координат, называемую экранной. При этом все объекты на экране имеют свою, локальную систему координат. Это показано на рисунке 4.5.

Мы можем определить объект в локальной системе координат, затем преобразовать эти координаты в мировые и, наконец, нарисовать наш объект в экранной систем координат. Выглядит это достаточно долгим занятием, но на

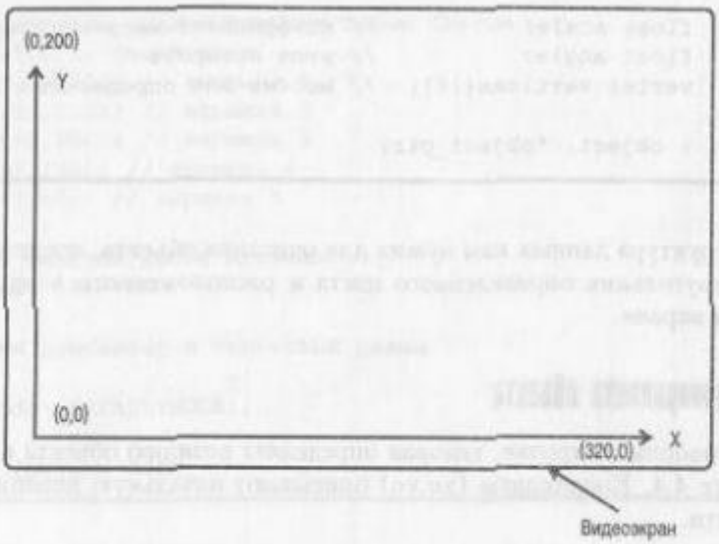


Рис. 4.6. *Переворачивание экрана для достижения полного соответствия с 1-м квадрантом картезианской системы.*

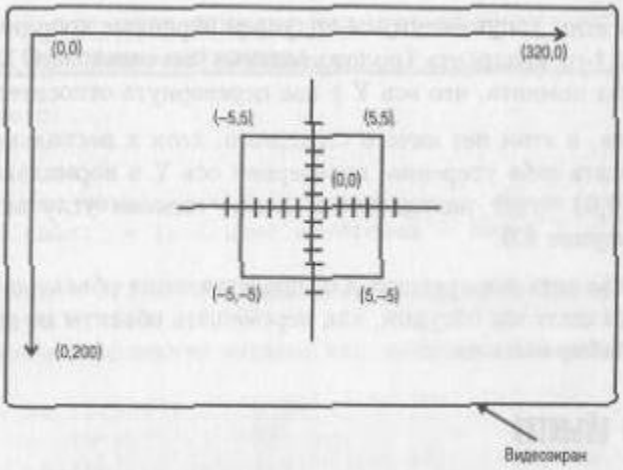


Рис. 4.5. *Локальная система координат объекта на экране.*

деле все оказывается несколько проще. Давайте договоримся, что у нас на компьютере мировые и экранные системы координат совпадают. Это значит, что:

- Точка (0,0) находится в левом верхнем углу экрана;
- При движении вправо увеличивается значение X-координаты;
- При перемещении вниз увеличивается Y-координата.

Благодаря этим допущениям мы получаем экранные координаты, похожие на координаты 1-го квадранта (положительные значения осей X и Y), но при этом надо всегда помнить, что ось Y у нас перевернута относительно экрана.

В принципе, в этом нет ничего страшного, хотя и несколько непривычно. Чтобы чувствовать себя уверенно, перевернем ось Y в нормальное положение. Тогда точка (0,0) будет находиться в левом нижнем углу экрана, как это показано на рисунке 4.6.

Теперь у нас есть все средства для представления объектов в компьютере. В следующем разделе мы обсудим, как перемещать объекты по экрану, поворачивать и масштабировать их.

Трансляция объектов

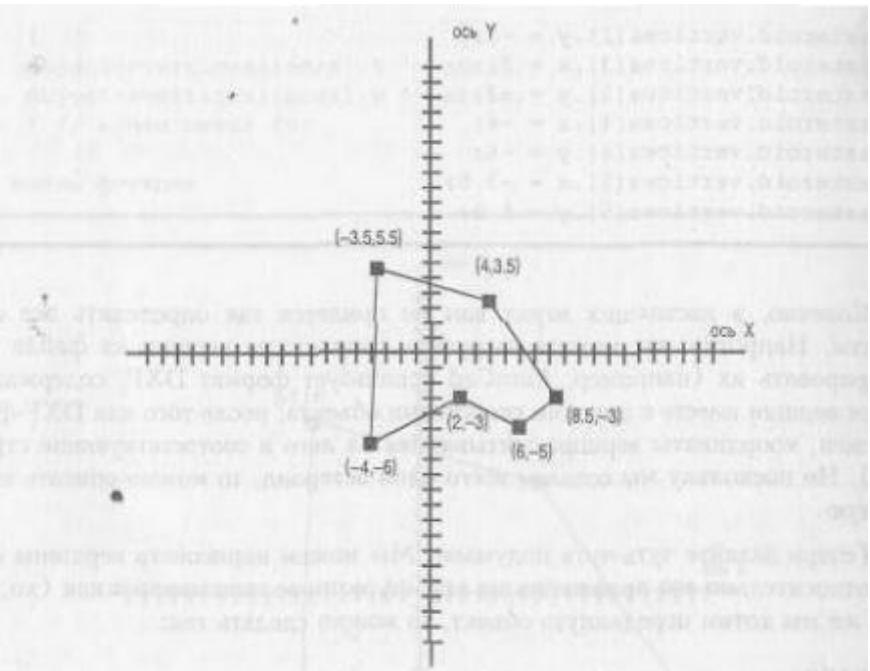


Рис. 4.7. *Многоугольник, определяющий вид нашего астероида.*

Трансляцией объекта будем называть его перемещение, при котором не меняется ни угол поворота, ни размер объекта. Давайте воспользуемся нашей структурой данных для определения конкретного объекта, с которым будем экспериментировать и в дальнейшем. К примеру, пусть это будет астероид. На рисунке 4.7 показан его внешний вид. Листинг 4.5 содержит фрагмент, описывающий наш астероид.

Листинг 4.5. Описание астероида.

```
Object asteroid;
// определим поля
asteroid.num_vertices = 6; //шести вершин будет достаточно
asteroid.color = 1; //цвет астероида - синий
asteroid.x0 = 320;   // поместить астероид в центр экрана
asteroid.y0 = 240;
//теперь задаем координаты вершин как смещения относительно точки x0, y0
asteroid.vertices[0].x = 4.0;
asteroid.vertices[0].y = 3.5;
asteroid.vertices[1].x = 8.5;
asteroid.vertices[1].y = -3.0;
asteroid.vertices[2].x = 6;
asteroid.vertices[2].y = -5;
asteroid.vertices[3],x = 2;
asteroid.vertices[3].y = -3;
asteroid.vertices[4].x = -4;
```

```
asteroid.vertices[4].y = -6;
asteroid.vertices[5].x = -3.5;
asteroid.vertices[5].y = 5.5;
```

Конечно, в настоящих играх вам не придется так определять все свои объекты. Напротив, вы можете загрузить координаты вершин из файла или сгенерировать их (например, AutoCad использует формат DXF, содержащий списки вершин вместе с другими свойствами объекта; после того как DXF-файл загружен, координаты вершин считываются из него в соответствующие структуры). Но поскольку мы создаем всего один астероид, то можно описать его и вручную.

Теперь давайте чуть-чуть подумаем. Мы можем нарисовать вершины объекта относительно его положения на экране, которое описывается как (xo,yo). Если же мы хотим передвинуть объект, то можно сделать так:

```
x0=x0+dx
y0=y0+dy
```

где dx и dy — это количество пикселей, на которое мы хотим переместить объект по оси X или Y.

Это все, что можно сказать о трансляции объектов. Теперь поговорим о масштабировании.

Масштабирование объектов

Масштабирование означает изменение размера объекта. Посмотрим для примера на рисунок 4.8. Астероид на нем в два раза больше, чем на рисунке 4.7. Это во многом объясняет принципы масштабирования. Все, что нам надо сделать, это умножить координаты каждой из вершин объекта на коэффициент масштабирования. Фрагмент кода в Листинге 4.6 показывает, как это делается для структуры объекта, определенной нами ранее.

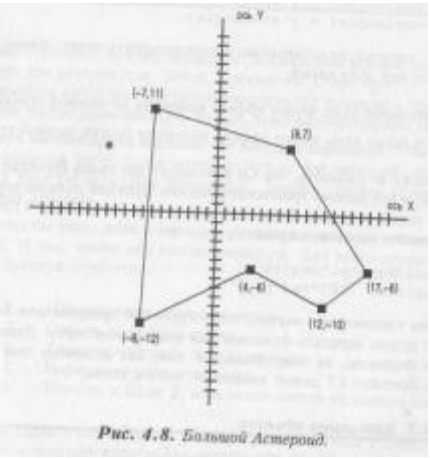


Рис. 4.8. Большой Астероид.

Листинг 4.6. Масштабирование астероида.

```
void Scale_Object(object_ptr object,float scale)
{
int index;
// для всех вершин масштабируем x- и y-компоненты
for (index = 0; index<object->num vertices; index++)
{
object->vertices [index].x *= scale;
object->vertlces[index].y *= scale;
} // конец цикла for
} // конец функции
```

Функция из Листинга 4.6 работает путем масштабирования координат каждой из вершин объекта. Если нам придет в голову увеличить наш объект «астероид»- в два раза, то нам потребуется написать следующее:

```
Scale_Object((object_ptr)&asteroid,2, 0};
```

С этим, вроде, все. Теперь мы уже готовы приступить к вращению объекта.

Вращение объектов

Для того чтобы вращать объект, мы должны повернуть его вокруг одной из координат. В двухмерной графике для этого обычно выбирается ось Z. Пока мы находимся в двухмерном мире, нас не беспокоит третье измерение - мы просто не придаем ему значения.

Если экран — это плоскость X-Y, то ось Z — это перпендикуляр к осям X и Y. Таким образом, если мы описываем наши объекты относительно двухмерного мира, то у нас появляется возможность вращать их относительно оси Z,

Следующие формулы позволяют вращать произвольную точку (X,Y) относительно оси Z:

```
new_x = x*cos(angle) - y*sin(angle) new_y = y*cos(angle) + y*sin(angle)
```

где angle — это угол, на который вы хотите повернуть точку. Кроме этого вам стоит помнить еще пару вещей:

- Положительные углы имеют эффект вращения по часовой стрелке;
- Отрицательные углы имеют эффект вращения против часовой стрелки.

Надо также не забывать, что Си использует для своих функций радианы, а не градусы, и все вызовы тригонометрических функций должны передавать в параметрах также радианы. Для того чтобы перевести радианы в градусы, мы должны написать простые макросы.

```
Deg_To_Rad(deg) {pi*deg/180;}
Rad_To_Deg(rad) {180*rad/pi;}
```

Другими словами, это значит, что в круге 360 градусов или 2xPi радиан. Теперь нам нужно написать функцию для вращения объекта. Давайте просто используем формулы, не задумываясь о том, как и почему они работают. Функция в Листинге 4.7 делает именно то, что мы хотим.

Листинг 4.7. Вращение объекта.

```
void Rotate_0bject(object__ptr object, float angle)
{
int index;
float x_new, y_new,cs, sn;
// сначала вычислим синус и косинус угла
cs = cos(angle) ;
sn = sin(angle);
// поворачиваем каждую вершину на угол angle
for (index=0; index<object->num_vertices; index++)
{
x_new = object->vertices [index].x*cs-object->vertices[index].y*sn;
y_new = object->vertices [index].y*cs+object->vertices[index].x*sn;
// изменяем исходные координаты.на расчетные
object->vertices[index].x = x_new;
object->vertices[index].y = y_new;
} // конец цикла for
} // конец функции
```

Думаю, что надо кое-что объяснить. Я вычисляю заранее значения синуса и косинуса для данного угла. Зачем, спросите вы. Ответ прост — для скорости. Ибо заниматься вычислениями тригонометрических функций в процессе работы программы можно позволить себе только, имея математический сопроцессор.

Теперь настало время написать что-нибудь посерьезней. Мне кажется, что надо бы написать что-то более экстравагантное, чем одинокий астероид. Пусть это будут хотя бы несколько астероидов. Давайте сначала спланируем наши дальнейшие действия.

Я хотел бы иметь поле астероидов различных размеров в количестве более 100 штук. И так, чтобы они могли вращаться. Для этого программа должна иметь следующую структуру:

Шаг 1. - Инициировать поле астероидов;

Шаг 2. - Стереть поле астероидов;

Шаг 3. - Трансформировать поле астероидов;

Шаг 4. - Нарисовать поле астероидов;

Шаг 5. - Перейти к Шагу 2, пока пользователь не нажмет на кнопку.

Чтобы сделать это проще, я добавил три новых поля к нашей структуре: одно для угла поворота и два - для скорости (целиком программа представлена в Листинге 4.8).

Листинг 4.8. Программа, которая рисует поле астероидов (FIELD.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ////////////////////////////////////
#include <stdio.h>
#include <graph.h>
#include <math.h>
// ОПРЕДЕЛЕНИЯ /////////////////////////////////////////
#define NUM_ASTEROIDS 10
#define ERASE 0
#define DRAW 1
// СТРУКТУРЫ ДАННЫХ ////////////////////////////////////
//определяем структуру "вершина"
typedef struct vertex_typ
{
float x,y; // координаты точки на плоскости
```

```
} vertex, *vertex__ptr;
// структура объекта
typedef struct object_typ
{
int num_vertices;     // количество вершин объекта
int color;            // цвет объекта
float xo,yo;          // позиция объекта
float x_velocity;     // скорость перемещения по осям X
float y_velocity;     // и Y
float scale;          // коэффициент масштабирования
float angle;          // угол поворота
vertex vertices[16];  // 16 вершин
}object, *object_ptr;
// Глобальные переменные ////////////////////////////
object asteroids[NUM_ASTEROIDS];
// Функции //////////////////////////////////////////
void Delay(int t)
{
// функция формирует некоторую временную задержку
float x = 1;
while(t—>0)
x=cos(x);
} // конец функции ////////////////////////////////////
void Scale_Object(object_ptr object,float scale)
{
int index;
// для всех вершин масштабируем координаты x и y
for (index = 0; index<object->num_vertices; index++)
{
object->vertices[index].x *= scale;
object->vertices[index].y *= scale;
}// end for index
// конец функции /////////////////////////////////////////
void Rotate_Object(object_ptr object, float angle)
{
int index;
float x_new, y_new,cs,sn;
// заранее вычислить синус и косинус
cs = cos(angle);
sn = sin(angle);
// поворачиваем каждую вершину на угол angle
for (index=0; index<object->num_vertices; index++)
{
x new = object->vertices[index].x * cs - object->vertices[index].y * sn;
y new = object->vertices[index].y * cs + object->vertices[index].x * sn;
object->vertices[index].x = x_new;
object->vertices[index].y = y_new;
} // конец цикла for
} // конец функции ////////////////////////////////////////////////
void Create_Field(void)
{
int index;
// формируем поле астероидов
for (index=0; index<NUM_ASTEROIDS; index++)
{
// заполнить все поля
asteroids[index].num_vertices = 6;
asteroids[index].color = 1 + rand() % 14; // всегда видимый
asteroids[index].xo    = 41 + rand() % 599;
asteroids[index].yo    = 41 + rand() % 439;
asteroids[index].x_velocity = -10 + rand() % 20;
asteroids[index].y_velocity = -10 + randO % 20;
asteroids[index].scale = (float)(rand() % 30) / 10;
asteroids[index].angle = (float) (-50+(float)(rand()%100))/100;
asteroids[index].vertices [0].x =4.0;
asteroids[index].vertices[0].y = 3.5;
asteroids[index].vertices[l].x=8.5;
```

```
asteroids[index].vertices[1].y = -3.0;
asteroids[index].vertices[2].x = 6;
asteroids[index].vertices[2].y = -5;
asteroids[index].vertices[3].x = 2;
asteroids[index].vertices[3].y =–3;
asteroids[index].vertices[4].x = -4;
asteroids[index].vertices[4].y = -6;
asteroids[index].vertices[5].x = -3.5;
asteroids[index].vertices[5].y =5.5;
// теперь масштабируем каждый астероид до нужного размера
Scale_Object((object_ptr)&asteroids [index],
asteroids[index].scale) ;
} // конец цикла for
} // конец функции ////////////////////////////////////////////////////
void Draw_Asteroids(int erase)
{
int index,vertex;
float xo,yo;
for (index=0; index<NUM_ASTEROIDS; index++)
{
// рисуем астероид
if (erase==ERASE)
_setcolor(0);
else
_setcolor(asteroids[index].color);
// получить позицию объекта
xo = asteroids[index].xo;
yo = asteroids[index].yo;
// перемещаемся к первой вершине
_moveto((int)(xo+asteroids[index].vertices[0].x),
(int)(yo+asteroids[index],vertices[0].y));
for (vertex=1; vertex<asteroids[index].num_vertices; vertex++)
{
_lineto((int)(xo+asteroids[index].vertlces[vertex].x),(int) (yo+asteroids[index].vertices
[vertex].y));
} // конец цикла for по вершинам
// замыкаем контур
_lineto((int)(xo+asteroids[index].vertices[0].x), (int)
(yo+asteroids[index].vertices[0].y));
} // конец цикла for по астероидам
} // конец
функции                                          ////////////////////////////////////////
////////////////////////////////////////////
void Translate_Asteroids()
{
int index;
for (index=0; index<NUM_ASTEROIDS; index++) {
// перемещаем текущий астероид
asteroids[index].xo += asteroids[index].x_velocity;
asteroids[index].yo += asteroids[index].y_velocity;
if (asteroids[index].xo > 600 || asteroids[index].xo < 40)
{
asteroids[index].x_velocity = -asteroids[index].x_velocity;
asteroids[index].xo += asteroids[index].x_velocity;
}
if (asteroids[index].yo > 440 ||  asteroids[index].yo < 40)
{
asteroids [index].y_velocity = -asteroids[index] .y_velocity;
asteroids[index].yo += asteroids[index].y_velocity;
}
} // конец цикла for } // конец функции
////////////////////////////////////////////////////
void Rotate_Asteroids(void)
{
int index;
for (index=0; index<NUM_ASTEROIDS; index++)
{
```

```
// вращаем текущий астероид
Rotate_0bject ((object_ptr) &asteroids [index],
asteroids[index].angle);
} // конец цикла for
} // конец функции
////////////////////////////////////////////////
void main(void) {
_setvideomode(_VRES16COLOR); //  640x480, 16 цветов
// инициализируем поле астероидов
Create_Field{) ;
while(!kbhit())
{ // очищаем поле
Draw_Asteroids(ERASE) ;
// изменяем поле
Rotate_Asteroids();
Translate_Asteroids();
// рисуем поле
Draw_Asteroids(DRAW) ;
// небольшая задержка
Delay(500);
} // конец цикла while
// устанавливаем текстовый режим
_setvideomode( DEFAULTMODE);
} // конец функции main
```

Набрав, откомпилировав и запустив программу из Листинга 4.8, вы увидите на экране поле астероидов вместе со множеством разноцветных камней, отскакивающих от границ экрана (определение факта столкновения будет подробнее излагаться далее в этой главе, а также в главе одиннадцатой, «Алгоритмы, структуры данных и методология видеоигр»)

Теперь нам надо еще кое-что обсудить:

- Первое, на что вы обратили внимание, это то, что образы слегка мелькают. Это связано с тем, что программа формирует изображение в тот же момент, когда происходит перерисовка экрана.
- Экран — это множество линий, которые рисуются на мониторе слева направо и сверху вниз вашей видеокартой. Проблема заключается в том, что мы не можем изменять видеобуфер, пока на экране что-то рисуется. В пятой главе, «Секреты VGA-карт», мы обсудим рисование всего экрана целиком в отдельный буфер с последующим перемещением его в видеопамять;
- Другая проблема, связанная с мерцанием, заключается в том, что мы используем графическую бибилотеку Microsoft C, которая не очень быстро работает. Вы должны понимать, что Microsoft не оптимизировал ее для высокой производительности и скорости работы;
- Программа использует числа с плавающей запятой, которые впоследствии будут заменены числами с фиксированной запятой;
- Вся программа совершенно неэффективна с точки зрения написания видеоигр. Все, что она делает, выполнено в классической, «книжной» манере. У разработчиков видеоигр есть правило номер 1: «Всегда есть способ сделать то, что кажется невозможным». Если б это было не так, то половина видеоигр вообще никогда не была бы написана, поскольку ПК не в состоянии обеспечить нужной производительности.

(Поверьте мне, DOOM — это лучший пример моим словам. Если бы я его не видел, то никогда не поверил бы, что такое возможно. Но, как мы знаем, DOOM существует, и это наилучший пример использования правильных методов для создания реального мира на компьютерном экране.)

Отсечения

Мне не хотелось бы углубляться в теорию, я собираюсь рассказать об отсечении просто для полноты изложения. Нам более интересны специальные способы отсечения, а не общая теория. Поэтому давайте изучим только основные концепции и их применение в наиболее общем виде.

Отсечение означает рисование только части видеообраза, ограниченного заранее заданными границами. Представьте себе объект, который перемещается по экрану. Дойдя до его границы, видимая часть объекта начинает уменьшаться и он постепенно уходит с экрана. Вы считаете это само собой разумеющимся, а программисту надо знать, какую часть образа он должен показать на: экране. Вычисления здесь могут быть сложными или простыми — все зависит от ситуации.

Отсечения могут производится на двух; «уровнях»:

- Уровень образа;
- Уровень объекта.

Отсечение области образов основывается на проверке каждой точки, кото рая может быть нарисована на экране в отсекаемой области. Например, если мы имеем квадратную область, которая соприкасается с границами экрана в режиме 13п (320x200), то мы не будем рисовать

точки, выходящие за границу. Точки, которые находятся внутри области и ограничены координатами 0-319 по оси X и 0-199 по оси Y, будут нарисованы, остальные - нет.

Поскольку все объекты, которые могут появляться на экране компьютера состоят из точек, то этот метод наиболее прост в реализации. Какие геометрические фигуры рисуются, в этом случае не имеет значения. Будь то линии, треугольники, битовые карты и еще что-нибудь - все они используют вызов функции рисования точки. Если функция, рисующая точку, определяет, что точка выходит за границы области, то данная точка не ставится.

Основная проблема, возникающая при разработке алгоритмов отсечения -это быстродействие. Давайте рассмотрим процесс рисования линии. Для этого достаточно узнать, где линия находится в отсекаемой области. Вы можете определить точки пересечения линии с границами области - к чему разбираться с каждой точкой, если можно рассматривать область целиком. Некоторые вычисления помогут определить, где линия будет разорвана. Это основа алгоритмов, позволяющих выполнять отсечения области объектов.

Алгоритмы отсечения области объектов рассматривают геометрические характеристики отсекаемых объектов и преобразуют их в цельные объекты, которые не надо разрезать. Эти новые объекты затем передаются в счетную часть программы и рисуются безотносительно к частям, которые оказались за пределами отсекаемой области. Сложность с алгоритмами области объектов состоит в том, что их реализация довольно сложна и зависит от типов рисуемых объектов. Отсечения простых линий из области объектов не так сложно, гораздо сложнее придумать эффективные алгоритмы для сложных объектов, состоящих из многоугольников.

К счастью для нас, наиболее сложный образ, который нам придется разрезать, это битовый образ (мы узнаем о том, как с ним работать в главе седьмой, «Продвинутая битовая графика и специальные эффекты»). Мы будем стремиться использовать эти алгоритмы всегда, когда это только возможно, поскольку они дают максимальную скорость. Мы не можем себе позволить добавлять в наши функции вывода точек всевозможные проверки, пока это не является абсолютной необходимостью.

Чтобы уверенно чувствовать себя в двухмерном мире, давайте поговорим на тему, которую большинство людей почему-то не любит. Речь пойдет о матрицах.

Матрицы

Матрицы не очень сложны для понимания и использования. Более того, они нужны дли написания быстрых преобразований и очень полезны для представления математических операций в компактной форме.

Матрица — это множество чисел, сгруппированных в колонки и столбцы. Здесь изображены две матрицы: Матрица A и Матрица B.

$$A = \begin{vmatrix} 1 & 0 & 2 \\ 3 & -1 & 4 \end{vmatrix} \qquad B = \begin{vmatrix} 3 & 5 & -1 \\ 2 & 7 & 0 \\ 4 & 3 & 3 \end{vmatrix}$$

Размерность 2x3 Размерность 3x3

Матрица A — это матрица 2x3 (то есть у нее две строки и три столбца), тогда как матрица B — это матрица 3x3. Мы можем получить доступ к элементу матрицы A, используя запись A[m,n], где m - это строка, а n - столбец. Элемент в левом верхнем углу матрицы A будет обозначаться A[0,0] и он равен, единице.

Произведение операций над матрицами

Вы можете производить большинство операций над матрицами так же, как вы оперируете и с нормальными числами. Например, вы можете их складывать или вычитать, соответственно складывая или вычитая каждый из компонентов.

Для примера, рассмотрим сложение двух матриц размерностью 2x3 - матрицы A и матрицы C:

$$A = \begin{vmatrix} 1 & 0 & 2 \\ 3 & -1 & 4 \end{vmatrix} \qquad C = \begin{vmatrix} 3 & 5 & -1 \\ 2 & 7 & 0 \end{vmatrix}$$

При сложении матриц A и C нужно складывать каждый из элементов m,n. Суммы элементов займут в результирующей матрице сответствующие места:

$$A + C = \begin{vmatrix} 1 & 0 & 2 \\ 3 & -1 & 4 \end{vmatrix} + \begin{vmatrix} 3 & 5 & -1 \\ 2 & 7 & 0 \end{vmatrix} = \begin{vmatrix} 4 & 5 & 1 \\ 5 & 6 & 4 \end{vmatrix}$$

Мы также можем умножить матрицу на скаляр k. Например, чтобы умножить матрицу A на 3, мы должны умножить на 3 каждый ее элемент:

$$3 \times A = 3 \times \begin{vmatrix} 1 & 0 & 2 \\ 3 & -1 & 4 \end{vmatrix} = \begin{vmatrix} 3 & 0 & 6 \\ 9 & -3 & 12 \end{vmatrix}$$

Теперь поговорим об умножении двух матриц. Эта операция немного отличается от умножения на скалярную величину. Вы должны запомнить несколько правил:

- Количество столбцов в первой матрице (n) должно быть равно количеству строк во второй

(также n). Это значит, что если размерность первой матрицы (mxn), то размерность второй матрицы должна быть (nxr). Два остальных измерения m и r могут быть любыми;

- Произведение матриц не коммутативно, то есть A x B не равно B x A.

Умножение матрицы mxn на матрицу nxr может быть описано алгоритмически следующим образом:

1. Для каждой строки первой матрицы:
   - Умножить строку на столбец другой матрицы поэлементно.
   - Сложить полученный результат;
2. Поместить результат в позицию [i,j] результирующей матрицы, где i - это строка первой матрицы, а j - столбец второй матрицы.

Для простоты посмотрим на рисунок 4.9:

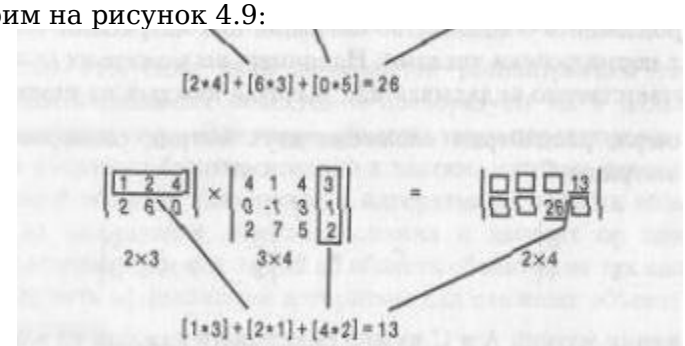


Рис. 4.9. *Результат матричного произведения.*

Мы можем это сделать намного проще, написав программу на Си. Давайте определим матрицу 3x3 и напишем функцию, умножающую матрицы. В Листинге 4.9 показана соответствующая программа.

Листинг 4.9. Определение и умножение двух матриц.

```
// общая структура матрицы
typedef sruct matrix_typ
{
float elem[3][3]; // место для хранения матрицы
} matrix, *matrix_ptr;
void Mat_Mult3X3 (matrix_ptr matrix_1, matrix_ptr matrix_2, matrix_ptr result)
{
index i,j,k;
for(i=0; i<3; i++)
{
for (j=0; j<3; j++)
{
result[i][j] = 0; // Инициализация элемента
for(k=0; k<3; k++)
{
result->elem[i][j]+=matrix_1->elem[i][k] * matrix_2->elem[k][j];
} // конец цикла по k
} // конец цикла по j
} // конец цикла по i
} // конец функции
```

Перед выходом из этой функции мы имеем результат, сохраненный в переменной result.

Единичная матрица

Прежде чем закончить говорить о матрицах, скажем еще об одной вещи: о единичной матрице- Не углубляясь в математические термины, я хочу сказать, что нам нужна такая матрица, умножая на которую мы получали бы исходную матрицу.

Говоря попросту, нам нужно иметь матрицу размерностью mxn, которую назовем матрицей I. Умножая на нее любую другую матрицу, мы должны получить исходную. Этой матрицей будет квадратная матрица, по главной диагонали которой записаны единицы, а все остальные элементы равны нулю:

$$A = \begin{vmatrix} 1 & 0 & 2 \\ 3 & -1 & 4 \\ 6 & -2 & 0 \end{vmatrix} \quad I = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Если мы умножим матрицу A на матрицу I,
то результатом будет исходная матрица A;
A x I = A

Использование матриц в играх

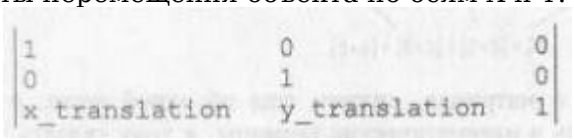
Мы уже достаточно поговорили о матрицах. Теперь посмотрим, зачем они нам нужны и как их использовать в играх. В программе из Листинга 4.8 мы создали поле астероидов, которые перемещались, изменяли свой размер и вращались. Если мы представим наши объекты как набор матриц, то сможем использовать умножение матриц для их трансформации.

Прелесть матриц состоит в том, что вы можете объединить все операции в одну матрицу, описывающую перемещение, масштабирование и вращение.

Помните, когда пишутся видеоигры, мы должны искать наиболее быстрый и эффективный алгоритм. Чем больше хитростей и трюков у нас в запасе, тем лучше. Теперь посмотрим на матрицы для перемещения, масштабирования и вращения.

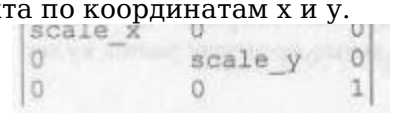
Главная матрица перемещений

Главной матрицей перемещений будем называть такую матрицу, в которой x_translation и y_translation - это коэффициенты перемещения объекта по осям X и Y. Вот как она выглядит:



Главная матрица масштабирования

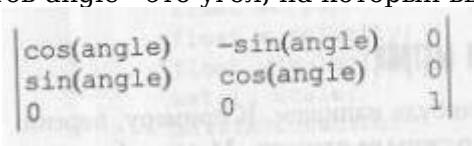
Главная матрица масштабирования - это такая матрица, в которой scale_x и scale_y - это коэффициенты масштабирования объекта по координатам x и y.



Такая матрица позволяет выполнять неоднородное масштабирование — мы можем задать один масштаб по оси X и другой — по Y. Таким образом, если мы хотим масштабировать объект однородно, то должны задать scale_x = scale_y.

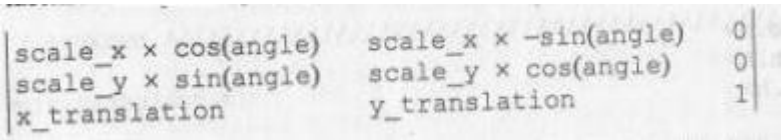
Главная матрица поворотов

В главной матрице поворотов angle - это угол, на который вы хотите повернуть, объект:



Общая матрица масштабирования, поворотов и перемещений

Наступает торжественный момент. Теперь мы возьмем матрицы перемещения, масштабирования и поворота и перемножим их (получим их конкатенацию), чтобы получить общую матрицу, реализующую все три функции сразу. Окончательно матрица будет выглядеть так:



Если вы теперь умножите вершины объекта на эту матрицу, то получите перемещенный, повернутый и масштабированный объект. Не слабо, а?

Компонент нормализации вершины

Я совсем упустил одну маленькую деталь, которую вы уже, наверное, заметили. «Как мы можем умножить вершину на матрицу размером 3x3?» Неплохой вопрос. Рад, что вы спросили. Чтобы выполнить это, мы должны изменить представление структуры вершин, добавив компонент нормализации. Компонент нормализации - это просто единица, добавленная в конец структуры, описывающей каждую вершину. Для этого нам надо чуть-чуть изменить исходные тексты в Листинге 4,4, в котором описаны вершины. Все это отражено в структуре данных в Листинге 4.10.

Листинг 4.10. Новая структура данных для вершин.

```
#define X_COMP 0
#define Y_COMP 1
#define N_COMP 2
typedef struct vertex_typ
```

```
{
float p[3];
) vertex, *vertex_ptr;
```

Как вы можете заметить, теперь мы превратили матрицу P, списывающую вершину, в массив. Наша матрица преобразований имеет размер 3x3, и для умножения нам надо иметь матрицу размером 1x3. Матрица P удовлетворяет этому условию.

Программа Астероиды с использованием матриц

Я уже устал от разговоров — давайте что-нибудь напишем. К примеру, перепишем нашу программу из Листинга 4.8. Вы должны ее помнить. Мы ее переделаем и используем в ней матрицы. Листинг 4.11 показывает эту программу. Она называется «Супер Астероиды».

Листинг 4.11. Супер Астероиды (FIELD.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ////////////////////////////////////////
#include <stdio.h>
#include <graph.h>
#include <math.h>
//ОПРЕДЕЛЕНИЯ /////////////////////////////////////////////
#define NUM__ASTEROIDS 10
#define ERASE 0
#define DRAW 1
#defineX_COMP 0
#define Y_COMP 1
#define N_COMP 2
// СТРУКТУРЫ ДАННЫХ ////////////////////////////////////
// новая структура, описывающая вершину
typedef struct vertex_typ
{
float p[3]; // единичная точка в двумерном пространстве
//и фактор нормализации
} vertex, *vertex_ptr;
// общая структура матрицы
typedef struct matrix_typ
{
float elem[3] [3]; // массив для хранения элементов
// матрицы
} matrix, *matrix_ptr;
// определение структуры "объект",.
typedef struct object_typ
{
int num_vertices;     // количество вершин в объекте
int color;            // цвет объекта
float xo,yo;          // позиция объекта
float x_velocity;     // скорости пермещения по осям X float y_velocity;
// и Y matrix scale;         // матрица масштабирования
matrix rotation;      // матрицы поворота и перемещения
vertex vertices[16];  // 16 вершин
} object, *object_ptr;
// ГЛОБАЛЬНЫЕ.ПЕРЕМЕННЫЕ ///////////////////////////////
object asteroids [NUM_ASTEROIDS] ;
// ФУНКЦИИ ////////////////////////////////////////////
void Delay(int t)
{
// функция выполняет некоторую задержку
float x = 1;
while(t—>0)
x=cos(x) ;
} // конец функции
//////////////////////////////////////////////////////
void Make_Identity(matrix_ptr i)
{
// функция формирует единичную матрицу
i->elem[0][0] = i->elem[l][1] = i->elem[2][2] = 1;
i->elem[0][1] = i->elem[l][0] = i->elem[l][2] = 0;
i->elem[2][0] = i->elem[01[2] = i->eiem[2][1] = 0;
} // конец функции ////////////////////////////////////////////////////
void Clear_Matrix(matrix_ptr m) {
```

```
// функция формирует нулевую матрицу
m->e1em[0][0] = m->elem[l][1] = m->e1em[2][2] = 0;
m->elem[0][1] = m->elem[l] [0] = m->e1em[1] [2] = 0;
m->elem[2][0] = m->elem[0] [2] = m->elem[2][1] = 0;
} // конец функции
//////////////////////////////////////////////////
void Mat_Mul (vertex_ptr v, matrix_ptr m)
{
 // функция выполняет умножение матрицы 1x3 на матрицу 3x3
// элемента. Для скорости каждое действие определяется "вручную",
// без использования циклов. Результат операции - матрица 1x3
float x new, y new;
x_new=v->p[0]*m->elem[0][0] + v->p[1]*m->elem[1][0] + m->elem[2][0];
y_new=v->p[0]*m->elem[0][1] + v->p[1]*m->elem[1][1] + m->elem[2][1];
// N_COMP - всегда единица
v->p[X_COMP] = x_new;
v->p[Y_COMP] = y_new;
} // конец функции
//////////////////////////////////////////////////
void Scale_Object_Mat(object_ptr obj)
{
// функция выполняет масштабирование объекта путем умножения на
// матрицу масштабирования
int index;
for (index=0; index<obj->num_vertices; index++)
{
Mat_Mul ((vertex_ptr) &obj->vertices [index],
(matrix_ptr)&obj->scale) ;
} // конец цикла for
} // конец функции
//////////////////////////////////////////////////
Rotate_Object_Mat(object_ptr obj)
{
// функция выполняет.поворот объекта путем умножения на
// матрицу поворота
int index;
for (index=0; index<obj->num_vertices; index++)
{
Mat_Mul((vertex_ptr)&obj->vertices[index],(matrix_ptr)&obj->rotation) ;
} // конец цикла for
} // конец функции //////////////////////////////////////////////////
void Create_Field(void)
{
int index;
float angle,c,s;

// эта функция создает поле астероидов
for (index=0; index<NUM_ASTEROIDS; index++)
{
asteroids[index].num vertices = 6;
asteroids [index] .color        = 1 + rand() % 14; //астероиды
// всегда видимы
asteroids[index].xo           = 41 + rand() % 599;
asteroids[index].yo           = 41 + rand() % 439;
asteroids [index].,x_velocity   = -10 +rand() % 20;
asteroids[index].y_velocity   = -10 + rand() % 20;
// очистить матрицу
Make_Identity((matrix_ptr)&asteroids[index].rotation) ;
// инициализировать матрицу вращений
angle = (float) (- 50 + (float) (rand ()\ % 100)) / 100;
c=cos(angle);
s=sin(angle);
asteroids[index].rotation.elem[0][0] = c;
asteroids[index].rotation.elem[0][1] = -s;
asteroids[index].rotation.elem[l][0] = s;
asteroids[index].rotation.elem[l][1] = c;
// формируем матрицу масштабирования
```

```
// очистить матрицу и установить значения коэффициентов
Make_Identity((matrix ptr)&asteroids[index].scale);
asteroids[index].scale.elem[0][0] = (float) (rand() % 30) / 10;
asteroids[index].scale.elem[1][1] = asteroids[index].scale.elem[0][0];
asteroids[index].vertices[0].p[X_COMP] = 4.0;
asteroids[index].vertices[0].p[Y_COMP] = 3.5;
asteroids[index].vertices[0].p[N_COMP] = l;
asteroids[index].vertices[1].p[X_COMP] = 8.5;
asteroids[index].vertices[l].p[Y_COMP) = -3.0;
asteroids[index].vertices[1].p[N_COMP] = l;
asteroids[index].vertices[2].p[X_COMP] = 6;
asteroids[index].vertices[2].p[Y_COMP] = -5;
asteroids[index].vertices[2].p[N_COMP] = l;
asteroids[index].vertices[3].p[X_COMP] = 2;
asteroids[index].vertices[3].p[Y_COMP] = -3;
asteroids[index].vertices[3].p[N_COMP] = l;
asteroids[index].vertices[4].p[X_COMP] = -4;
asteroids[index].vertices[4].p[Y_COMP] = -6;
asteroids[index].vertices[4].p[N_COMP] = 1;
asteroids[index].vertices[5].p[X_COMP] = -3.5;
asteroids[index].vertices[5].p[Y_COMP] = 5.5;
asteroids[index],vertices[5].p[N_COMP] = 1;
// теперь масштабировать астероиды
Scale_Object_Mat((object_ptr)&asteroids[index]);
} // конец цикла for
} // конец функции
//////////////////////////////////////////////////
void Draw Asteroids (int erase)
{
int index,vertex;
float xo,yo;
// эта функция в зависимости от переданного флага рисует или стирает
// астероиды
for (index=0; index<NUM_ASTEROIDS; index++)
{
// нарисовать астероид
if (erase==ERASE)
_setcolor(0);
else
_setcolor(asteroids[index].color);
// получаем позицию объекта
xo = asteroids[index].xo;
yo = asteroids [index].yo;
// перемещаемся в позицию первой вершины астероида
_moveto((int)(xo+asteroids[index]-vertices[O].p[X_COMP]), (int)
(yo+asteroids[indexl.vertices[0].p[Y_COMP])) ;
for (vertex=l; vertex<asteroids[index].num_vertices; vertex++)
{
_lineto((int)(xo+asteroids[index].vertices[vertex].p[X_COMP]), (int)
(yo+asteroids[index].vertices[vertex].p[Y_COMP])) ;
} // конец цикла по вершинам
_lineto((int)(xo+asteroids[index].vertices[0],p[X_COMP]), (int)
(yo+asteroids[index].vertices[0].p[Y_COMP]));
} // замкнуть контур объекта
} // конец функции
//////////////////////////////////////////////////
void Translate_Asteroids(void)
{
// функция перемещает астероиды
int index;
for (index=0; index<NUM_ASTEROIDS; index++)
{
// перемещать текущий астероид
asteroids[index].xo += asteroids[index].x velocity;
asteroids[index].yo += asteroids[index].y_velocity;
// проверка на выход за границы экрана
if (asteroids[index].xo > 600 || asteroids[index].xo < 40)
```

```
{
asteroids[index].x_velocity = -asteroids[index].x_velocity;
asteroids[index].xo += asteroids[index].x_velocity;
      }
if (asteroids[index].yo > 440 1) asteroids[index].yo < 40)
{
asteroids[index].y_velocity = -asteroids[index].y_velocity;
asteroids[index].yo += asteroids[index].y_velocity;
)
} // конец цикла for
} // конец функции
//////////////////////////////////////////////////////
void Rotate__Asteroids()
{
int index;
for (index=0; index<NUM_ASTEROIDS; index++)
{
// вращать текущий астероид
Rotate_Object_Mat((object_ptr)&asteroids[index]);
} // конец цикла for
} // конец функции
//////////////////////////////////////////////////////
void main(void)
{
// перевести компьютер в графический режим
_setvideomode(_VRES16COLOR); // 640x480, 16 цветов
// инициализация
Create_Field();
while(!kbhit())
{
// стереть поле
Draw_Asteroids(ERASE) ;
// преобразовать поле Rotate_Asteroids();
Translate_Asteroids() ;
// нарисовать поле
Draw_Asteroids(DRAW);
// немного подождем...
Delay(500);
)
// перевести компьютер в текстовый режим
_setvideomode(_DEFAULTMODE);
} // конец функции
```

Вам потребуется время, чтобы изучить эту программу. Уделите внимание способу, которым астероиды масштабируются и поворачиваются. Если вы сравните время исполнения программ из Листингов 4.8 и 4.11, то не найдете никакой разницы. Если же вы потратите время и поместите все повороты, масштабирование и перемещение в одну матрицу, то программа из Листинга 4,11 будет работать значительно быстрее.

Итак, умножение матриц особенно эффективно, когда вы производите множественные трансформации объекта с помощью единой, заранее подготовленной матрицы, включающей в себя все операции преобразования над этим объектом.

К вашему сведению

Запомните, что все это мы делаем для того, чтобы научиться писать видеоигры, особенно трехмерные. И делаем мы это постепенно. Если я вам сразу покажу, как писать игры типа Wolfenstein 3-D или DOOM, то мы многое потеряем в тактике и философии разработки и создания игр.

Если же вы настоящий разработчик игр, то уже в данный момент должны думать,о том, как переделать те программы, которые я дал, в настоящую игру. Не беспокойтесь! Мне и в голову не приходит сказать: «В качестве упражнения перепишите программу из Листинга 4.11 в настоящую игру». Мы напишем игру вместе, но чуть позже. Лучше поговорим о том, какие изменения нам надо внести, чтобы игра получилась.

Если вы хотите узнать, как я бы переделывал Листинг 4.8 в настоящую игру, то я бы сделал следующее:

- Оснастил программу всем необходимым для выполнения миссии. Ведь нужно иметь маневренный космический корабль, несколько ракет для уничтожения астероидов, а чтобы игра приняла «товарный» вид, нужна система подсчета очков, заставка и, конечно,

возможность выхода из программы. Кроме перечисленного потребуется создать систему оповещения о столкновениях с астероидами (такая система будет рассмотрена чуть позже, в этой же главе);

- Далее я подготовил бы свой корабль. Для этого описал бы объект, который позаимствовал, например, из игры Star Trek;
- Написал бы подпрограмму для рисования и уничтожения корабля (типа функции DrawAsteroids из Листинга 4.11);
- Разработал бы функции, позволяющие игроку управлять кораблем;
- Написал бы функции, которые, в зависимости от состояния клавиатуры, вращали корабль по или против часовой стрелки. Это я сделал бы, изменяя величину angles в структуре, описывающей объект корабля;
- Потом мне понадобился бы способ передвижения корабля. Для этого я бы изменял значения x_velocity и y_velocity в соответствующих функциях:

```
x_veiocity = cos(angle)*speed
y_velocity = sin(angle)*speed
```

- где скорость изменялась бы при нажатии на определенные клавиши;
- Чтобы ввести корабль в игру, я поместил бы вызов функции стирания всех графических объектов - корабля и астероидов - в одно и то же место. Затем поместил бы функцию движения, которая опрашивает клавиатуру, в середину программы (между вызовом функций стирания и рисования);
- В самый конец цикла вставил бы функцию рисования корабля.

Вот, собственно, и все. Выглядит не очень сложно. Посмотрим, что дальше.

Основы контроля столкновений

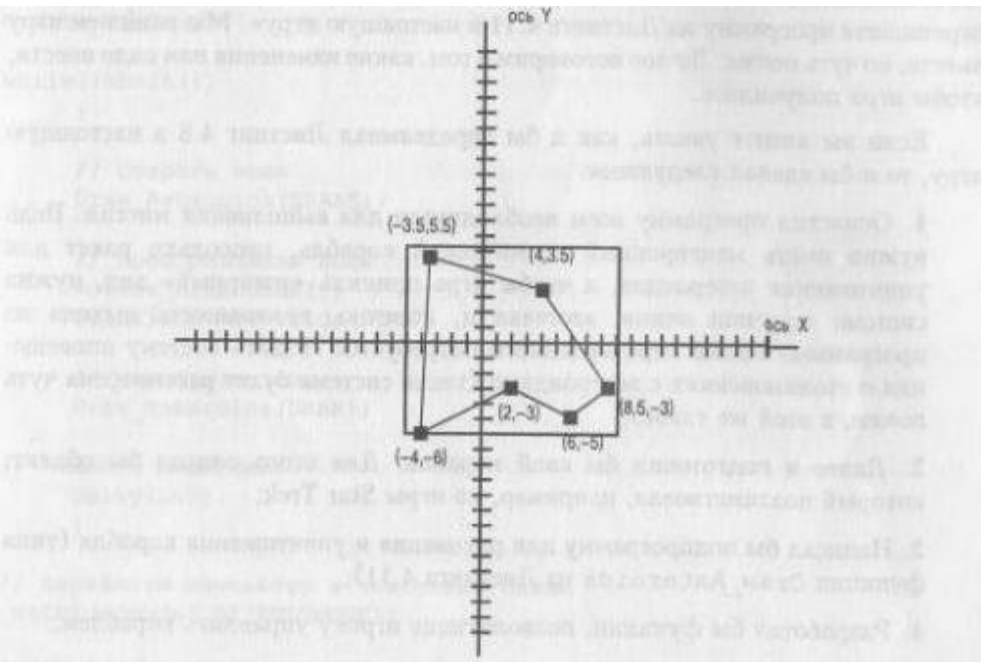


Рис. 4.10. Описывающий прямоугольник вокруг астероида для выявления столкновений.

Давайте забудем про оружие и посмотрим, что произойдет, если нас заденет астероид (я слышал, они очень твердые). Для этого нам надо создать систему контроля столкновений.

Это значит, что нам надо сделать зримым соударение космического корабля с астероидом. Правильнее всего будет поступить так:

1. Забыть о реальных границах астероида и корабля и создать виртуальную граничную область вокруг каждого объекта исключительно для тестовых целей. Не надо ее рисовать, достаточно, чтобы она присутствовала внутри программы. Рисунок 4.10 показывает, что я имею в виду.
2. Если мы увидим, что граничная область нашего судна вошла в граничную область астероида, мы можем сказать, что наш корабль уничтожен. Это довольно просто сделать: достаточно сравнить каждую вершину области с вершиной другой области. Если есть вхождение, значит произошло столкновение.
3. Теперь, когда столкновение зафиксировано, нужно произвести на экране какой-нибудь красочный эффект: 500-тонный корабль с двигателем на антивеществе врезается в астероид. Например, вспышки случайных точек. Это будет неплохо смотреться.
4. Подсчет очков прост: создайте переменную score и печатайте ее на экране каждый раз, проходя через главный цикл. Вы можете использовать функцию _outtext () из библиотеки Си.

Кроме этого, нам надо набирать очки. Значит, нужно иметь оружие. Например, корабль в игре может использовать фотонные торпеды — маленькие точки. Чтобы их получить, нам надо:

1. Назначить одну из клавиш (лучше Пробел**)** для стрельбы.
2. Когда будет обнаружено нажатие на клавишу, определить несколько переменных, в

которых запомнить направление движения корабля (miss_xv и miss_yv).

3. Затем запомнить в двух переменных исходную позицию ракеты, то есть координаты центра корабля (miss_x и miss_y).
4. Присвоить переменной life значение 100.
5. В цикле начать перемещение ракеты параллельно курсу корабля и каждый раз сравнивать координаты торпеды с границами астероидов. Если столкновение обнаружено, увеличить счетчик очков и удалить астероид.
6. Уменьшить значение переменной life, и если она стала равна нулю, то уничтожить ракету и дать возможность игроку выстрелить следующий раз.

ИТОГ

Эта глава могла бы оказаться в сотни раз больше. У меня действительно была задача провести краткий курс двухмерной графики. Мы узнали много новых концепций и терминов. Теперь мы знаем, как производить простые преобразования объектов и точек на плоскости. Мы также узнали о матрицах (даже несколько больше, чем нужно) и как они могут быть использованы в компьютерной графике и видеоиграх для упрощения геометрических преобразований и придания им более компактной формы. В конце мы даже увидели поле астероидов на экране. Я уверен, что у вас есть много вопросов. Наверное, даже больше, чем в начале изучения этой книги. Я постараюсь ответить на них в дальнейшем. Но прошу вас помнить, что не существует простых и быстрых правил для написания игр. Это не сортировка массива. Нам, как программистам, нужен набор различных трюков для реализации наших идей, а посему продолжим работу.

СЕКРЕТЫ VGA-КАРТ

Когда мы пишем видеоигры, то используем компьютер для создания миров и VGA-карту для рисования разнообразных объектов. В этом случае VGA-карта является «связующей нитью» между компьютером и экраном дисплея. В данной главе мы поговорим о следующем:

- Обзор VGA-карт;
- 256-цветный режим;
- Конфигурация видеопамяти;
- Таблицы цветов;
- Переопределение цветовой палитры;
- Цветовое вращение;
- Рисование точек;
- Рисование линий;
- Файлы PCX;
- Вывод битовых образов (бит-блиттинг);
- Спрайты;
- Вывод текста
- Дублирующее буферизирование;
- Синхронизация с вертикальной разверткой;
- Игра Tombstone

Обзор VGA-карт

VGA-карта это плата, которая генерирует видеосигнал, передаваемый на дисплей. Этот видеосигнал состоит из серии импульсов, определяющих цвет и интенсивность каждой точки на экране. Не будем вдаваться в подробности о том, как он образуется или как карта VGA создает все временные задержки для корректной работы с дисплеем. У нас есть единственный вопрос: как мы можем поместить пиксель нужного цвета на экран?

Нарисовать одиночную точку на экране было бы достаточно непростой задачей, и нам пришлось бы проделать для этого множество разных вещей. Но, проделав это один раз, мы сможем в дальнейшем вызывать уже написанные функции и никогда больше об этом не вспоминать.

Как вы знаете, VGA-карта поддерживает много различных графических режимов и может быть перепрограммирована для получения совершенно фантастических разрешений (например, я создал режим 320x400, 256 цветов путем прямого перепрограммирования временных регистров VGA-карты). Такая функциональность стала возможной благодаря архитектуре VGA-карты. Карта VGA имеет ряд регистров, которые можно по-разному программировать для решения различных задач. В любом случае, программирование регистров VGA — это тема для отдельной книги. Если вы этим огорчены, не отчаивайтесь, существует множество книг на эту тему. В моей маленькой домашней библиотеке более 30 книг посвящены VGA-картам. Если вы хотите узнать больше про секреты VGA, то, я уверен, у вас не возникнет проблем с источниками информации.

Один из путей приблизиться к секретам VGA-карты, это рассматривать ее как систему, поддерживающую только один режим; например, режим, который мы и будем применять при создании наших игр, дающий разрешение 320x200 точек, 256 цветов, называемый режимом 13h. Детали мы обсудим позже в этой главе.

Используя только один из всевозможных графических режимов, мы становимся более независимыми от различий карт VGA. Почему? Наши игры будут работать только на VGA и SVGA и нам не потребуется поддерживать EGA, монохромные и другие карты. Часто разработчики видеоигр пишут десятки драйверов для поддержки разных карт и мониторов- Если мы сосредоточимся на одном режиме, то сможем освободиться от излишней общности и будем писать код только один раз.

Обычно, когда вы пишете програм', то стараетесь сделать ее достаточно универсальной. К сожалению, это не относится к видеоиграм для ПК. Игровые программы настолько насыщены графикой и вычислениями, что нам приходится все переписывать заново для каждой новой игры. Ведь мы не делаем текстовых процессоров, а создаем новые миры, и не можем повторяться, используя библиотеки от старых игр. Я думаю, это правильно.

Мы будем использовать VGA в режиме 13h. Но как этот режим работает? Прочитав следующий раздел, вы найдете ответ на этот вопрос.

256-цветный режим

В режиме 13h (или 19 в десятичной нотации) экран состоит из 320 пикселей по горизонтали и 200 — по вертикали. В нем поддерживается одновременное отображение до 256 цветов.

Теперь мы приближаемся к пониманию истинной ценности режима 13h. Он является наиболее простым в программировании. Как вы заметили, все графические режимы (CGA, EGA, VGA, SVGA, XGA и другие) используют некоторую область памяти для представления битовой карты, выводимой на экран. Этот раздел памяти называется видеобуфером. Для режима 13h и VGA-карт видеобуфер начинается с адреса A000:0000 и простирается до адреса AOOO;F9FF. Если вы владеете шестнадцатеричной арифметикой, то легко увидите, что размер буфера равен 64000 байт. Умножив 320 на 200, мы получим тот же результат. Это значит, что каждый пиксель в режиме 13h представлен одним байтом. Этот замечательный факт заставляет меня любить VGA-карту. На рисунке 5.1 показано строение видеобуфера VGA.

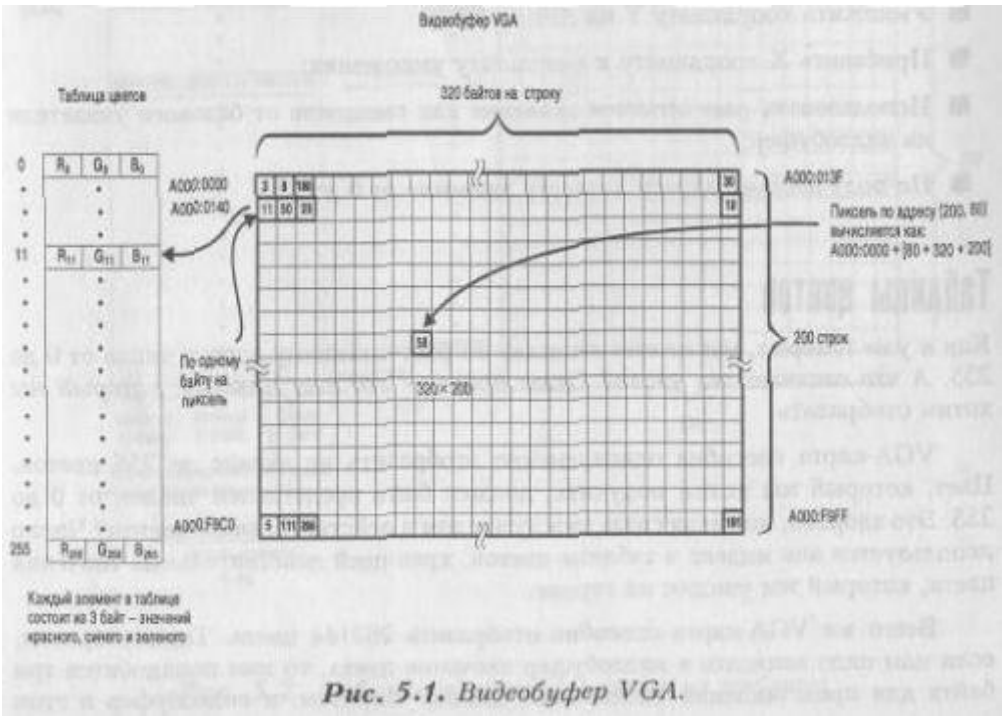


Рис. 5.1. Видеобуфер VGA.

Позже в этой главе мы узнаем, как адресовать видеобуфер и записывать в него пиксели. Сейчас же я хочу поговорить о другом. Может возникнуть вопрос: «А не является ли экран 320x200 слишком маленьким?» Ответом будет: «И да, и нет». Сегодня разрешение 320x200 устарело, но благодаря 256 цветам и правильному рендерингу оно и по сию пору выглядит восхитительно. Эти плюсы заставляют нас забыть о низком разрешении. Кстати, игрок никогда не должен даже подозревать, что программа работает в режиме с низкой разрешающей способностью экрана.

Конфигурация видеопамяти

Давайте поговорим о записи в видеобуфер. Как вы можете видеть из рисунка 5.1, видеобуфер начинается с адреса A000:0000 и заканчивается по адресу AOOO:FF9F. Поскольку каждый пиксель представлен одним байтом, то адресация видеобуфера довольно проста. Все, что нам нужно сделать, это определить указатель на видеобуфер, после чего можно записывать пиксели, используя этот указатель. И все!

Видеобуфер — это одна большая, непрерывная область памяти или массив байтов. Мы знаем, что он состоит из 320 байтов в каждой строке и таких строк насчитывается 200. Таким образом, чтобы нарисовать пиксель с координатами (x,y), нам надо:

- Умножить координату Y на 320;
- Прибавить X-координату к результату умножения;
- Использовать рассчитанное значение как смещение от базового указателя на видеобуфер;
- По полученному адресу записать значение от 0 до 255.

Таблицы цветов

Как я уже говорил, мы можем записать по рассчитанному адресу число от 0 до 255. А что означает это число? Ответ прост - это цвет пикселя, который мы хотим отобразить.

VGA-карта способна одновременно отобразить на экране до 256 цветов. Цвет, который мы хотим получить, должен быть представлен числом от 0 до 255. Это здорово, но какая связь между числом и действительным цветом? Число используется как индекс в таблице цветов, хранящей действительные значения цвета, который мы увидим на экране.

Всего же VGA-карта способна отобразить 262144 цвета. Таким образом, если нам надо записать в видеобуфер значение цвета, то нам понадобится три байта для представления числа такой длины. Впрочем, и видеобуфер в этом случае будет просто огромен. Поэтому, создатели карты предусмотрели возможность переадресации графического адаптера.

Переадресация означает, что одна числовая величина используется в качестве адреса другого значения (примерно, как именованный указатель в Си). Вместо одновременного воспроизведения всех 262144 цветов, разработчики VGA-карт дали возможность использовать

подмножество из 256 цветов. В результате VGA-карта имеет таблицу отображения цветов, включающую 256 значений. Каждое из этих значений состоит из 256 элементов размером в один байт, содержащих значения красного, синего и зеленого для выбранного цвета (помните, красный, зеленый и синий в комбинации могут образовывать любой цвет.)

Таблица цветов состоит из 768 байт (3x256). Например, когда карта считывает из видеобуфера число 72, то она проверяет адрес 72 в таблице цветов. Адрес 72 находится по смещению 72x3 от начала таблицы, поскольку каждое из значений занимает три байта. Затем значения зеленого, красного и синего считываются из таблицы и используются как значения сигналов. Давайте рассмотрим рисунок 5.2 для более подробного ознакомления.

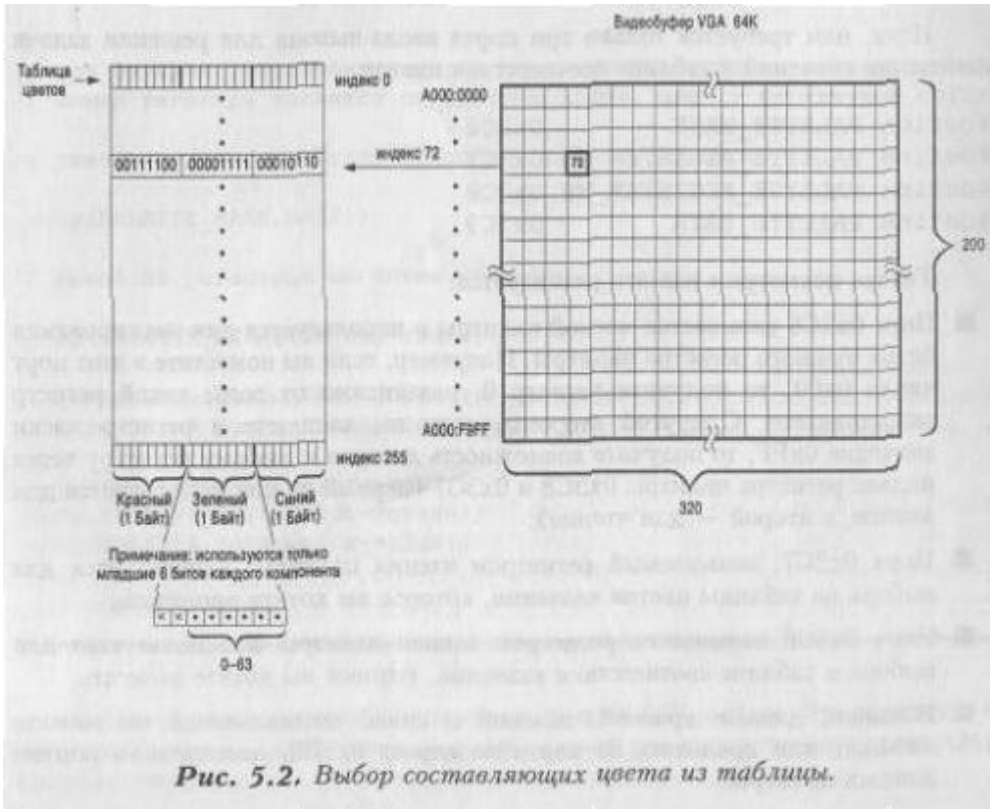


Рис. 5.2. *Выбор составляющих цвета из таблицы.*

Было бы прекрасно, если б мы имели прямой доступ к таблице соответствия, как к обычной памяти. Но, к сожалению, регистры цвета доступны только через порты ввода-вывода VGA-карты. На самом деле это плохо, поскольку весьма усложняет нам жизнь. Теперь нам предстоит узнать, как менять значения в таблице цветов.

Переопределение цветовой палитры

Таблица цветов организована в VGA-карте как регистровый файл. (Я использовал слово регистр, чтобы обозначить значение в таблице соответствия. Каждый регистр палитры — это 24 бита.) Для доступа к значению мы должны произвести некоторые действия. Мы не можем просто сказать; «Изменить компонент красного для значения 123». Мы должны модифицировать все три составляющих цвета, который хотим изменить.

Хотя каждое значение состоит из трех байтов (один для каждой из составляющих), только первые шесть битов каждого байта используются для обозначения цвета. Существует 64 оттенка для каждого цвета, или 2 в 18-й степени различных цветов (это и есть общее количество цветов — 262144). Таким образом, если вы поместите значение, превышающее размер в шесть битов (или 63), то можете нарушить все компоненты, но не изменить цвет.

Итак, нам требуется только три порта ввода-вывода для решения задачи изменения значений в таблице соответствия цветов.

```
#define PALETTE_MASK        0x3C6
#define PALETTE_REGISTER_RD 0x3C7
#define PALETTE_REGISTER_WR 0x3C8
#define PALETTE_DATA        0x3C9
```

Теперь посмотрим как это реализуется:

- Порт 0x3Cб называется маской палитры и используется для маскирования битов нужного регистра палитры. Например, если вы поместите в этот порт число 0x00, то получите регистр 0, независимо от того, какой регистр запрашиваете. С другой стороны, если вы запишете в регистр маски значение 0xFF, то получите возможность доступа к любому регистру через индекс регистра палитры 0x3C8 и 0x3C7 (первый из них используется для записи, а второй — для чтения);
- Порт 0x3C7, называемый регистром чтения палитры, используется для выбора из таблицы цветов значения, которое вы хотите прочитать;

- Порт 0x3C8 называется регистром записи палитры и используется для выбора в таблице соответствия значения, которое вы хотите записать;
- Наконец, данные красной, зеленой и синей составляющей вы можете записать или прочитать из порта по адресу 0x3C9, называемого портом данных палитры.

Вы можете спросить: «А как мы прочитаем из одного порта три байта?» На самом деле вы можете прочитать их по очереди. После того как вы выберете необходимый регистр (значение таблицы цветов, к которому вам нужен доступ), то первый записанный в регистр палитры байт будет соответствовать значению красного цвета. Второй байт задаст значение зеленого цвета, ну а третий — синего. Когда вы будете читать, это правило будет так же верно, но. в отличие от записи трех байтов в каждый момент чтения вы будете получать следующий компонент считываемого значения выбранного регистра. Для записи в регистр палитры вы должны:

- Выбрать регистр, который хотите изменить;
- Произвести три записи в порт регистра данных.

Когда вы читаете или пишете в регистр цвета, не забывайте каждый раз предварительно изменять значение маски на OxFF. Листинг 5.1 показывает код, содержащий эти шаги.

Листинг 5.1. Запись в регистр палитры.

```
void Set_Palette_Register(int index, RGB_color_ptr color)
{
// эта функция устанавливает один из элементов таблицы цветов.
// номер регистра задается параметром index, цвет - структурой color
// указываем, что мы будем обновлять регистр палитры
_outp(PALETTE_MASK,Oxff) ;
// какой из регистров мы хотим обновить
_outp(PALETTE_REGISTER_WR, index);
// теперь обновляем RGB. Обратите внимание,
// что каждый раз используется один и тот же порт
_outp(PALETTE_DATA,color->red) ;
_outp(PALETTE_DATA,color->green) ;
_outp(PALETTE_DATA,color->blue) ;
} // конец функции
```

Помните, что я использую структуру данных RGB_color. Это структура с тремя полями, каждое из которых предназначено для определенного цвета. Она выглядит так:

```
typedef struct RGB_color_typ
{
unsigned char red;    // красный компонент 0-63
unsigned char green; // зеленый компонент 0-63
unsigned char blue;   // синий компонент 0-63
} RGB_color, *RGB_color_ptr;
```

Все походит на то, что следующей операцией должно стать чтение из регистра. Мы сделаем то же самое, что и в Set_Palette_Register, только вместо записи в порт палитры будем читать из него и пересылать полученные значения в структуру RGB color. Листинг 5.2 содержит необходимый для этого код.

Листинг 5.2 Чтение регистра палитры.

```
void Get_Palette_Register(int index, RGB_color__ptr color)
{
// функция читает регистры цвета и помещает полученные значения
// в поля структуры color
// установить маску регистра палитры
_outp(PALETTE_MASK,0xff);
// сообщаем VGA, какой из регистров мы будем читать
_outp(PALETTE_REGISTER_RD, index);
// читаем данные
color->red   = _inp(PALETTE_DATA);
color->green = _inp(PALETTE_DATA);
color->blue = _inp(PALETTE_DATA);
} конец функции
```

Теперь, когда мы знаем, как читать и писать в регистр палитры, почему бы нам не создать функцию для получения новой цветовой палитры? Неплохая идея! Напишем для этого функцию, которая строит палитру, имеющую 64 оттенка серого, красного и голубого. Листинг 5.3 содержит ее код.

Листинг 5.3. Создание новой цветовой палитры.

```
void Create_Cool_ Palette(void)
```

```
{
// эта функция создает новую палитру, содержащую по 64 оттенка
// серого, красного, зеленого и синего цветов
RGB_color color;
int index;
// в цикле последовательно создаем цвета и меняем значения регистров
for (index=0; index < 64; index++)
{ // это оттенки серого
color.red   = index;
color.green = index;
color.blue = index;
Set_Palette_Register(index, (RGB_color_ptr)&color);
// это оттенки красного
color.red   = index;
color.green = 0;
color.blue = 0;
Set_Palette_Register(index+64, (RGB_color_ptr)&color) ;
// это оттенки зеленого color.red   = 0;
color.green = index;
color.blue = 0;
Set_Palette_Register(index+128, (RGB_color_ptr)&color) ;
// это оттенки синего
color.red   = 0;
color.green = 0;
color.blue = index;
Set_Palette_Register(index+192, (RGB_color_ptr)&color);
} // конец цикла for
} // конец функции
```

Наличие возможности изменения цветовой палитры позволяет нам создавать различные интересные эффекты освещения и анимации в наших играх. (Например, как сделаны цветовые эффекты в DOOM'е? Часть цветовой палитры изменяется «на лету» во время игры.) Так достигаются эффекты освещения и стрельбы. Это особенно хорошо для реализации эффекта «отключения света».

Вы должны принять на веру работоспособность функции из Листинга 5,3, поскольку у нас нет возможности проверить ее. Прежде мы должны были бы разработать эффективные функции для рисования точки и линии, но сейчас я хочу поговорить о технике, применение которой мы вскоре найдем.

Цветовая ротация

Когда я купил свой первый компьютер в 1978 году (хотя я до сих пор уверен, что это был божий дар) я был глубоко поражен компьютерной графикой и видеоиграми. Компьютер назывался Atari 800. В то время это было пределом возможностей. Один из интересных эффектов, который он поддерживал, назывался цветовой ротацией. Цветовая ротация может быть выполнена только на компьютерах, имеющих таблицу преобразования цветов. Как известно, изображение рисуется на экране компьютера. Например, пусть это будет водопад. Изображение водопада состоит (в нашем случае) из 16 оттенков синего цвета, каждый из которых — это число, ссылающееся в таблице преобразования на соответствующее значение цвета. Мы получаем, что водопад содержит 16 оттенков синего, находящихся в регистрах цвета с адресами от 100 до 115.

Теперь представьте, что мы берем одно из значений и сдвигаем его в следующий регистр и так далее до 115-го, содержимое которого переносим в 100-й регистр. Что произойдет? Правильно, возникает ощущение движения. Уже в 70-х годах это было возможно на процессорах 6502 с тактовой частотой 1.79МГц.

Мы применим эту технику позже. Есть куча классных вещей, которые можно сделать, используя этот прием, тем более, что в ПК сейчас целых 256 регистров. А пока постарайтесь просто запомнить этот трюк.

Рисование точки

Однажды я сказал: «Дайте мне адрес видеобуфера и я переверну экран. ..». Это весьма правдивое высказывание. Во всех системах с отображением адресов видеопамяти на область адресов памяти обычной, как это делается в ПК; рендеринг был бы более простой и доступной вещью, если бы видеобуфер имел хоть каплю логики в своей организации. Собственно, организация и логика есть: видеобуфер - это один большой массив. И все.

Как мы узнали чуть раньше, для рисования точки нам достаточно вычислить адрес смещения относительно начала видеобуфера (A000:0000) и записать байт, отображающий ее цвет. Больше ничего делать не надо. Листинг 5.4 содержит фрагмент кода, который рисует точку определенного цвета с координатами X и Y.

Листинг 5.4. Рисование точки в позиции (x,y).

```
void Plot_Pixel(int x, int y, unsigned char color)
```

```
{
// эта функция отображает точку выбранного цвета. Каждая строка
// занимает 320 байт, поэтому для вычисления адреса надо умножить Y
// на 320 и прибавить значение X
video_buffer[y*320+x]=color;
} // конец функции
```

Итак, рисование точки довольно просто. Я думал, все окажется сложнее, и поэтому так много написал об этом, но то, что мы использовали режим 13h, значительно упростило дело. Функция Plot_Pixel получилась простой, это всего одна строка кода. Тем не менее, давайте попробуем ее оптимизировать.

В книге есть целая глава, посвященная оптимизации, но эта единственная строка, содержащая операцию умножения, меня сводит с ума. Давайте избавимся от умножения. Кстати, возьмем за правило избегать операций умножения и вообще, откажемся от действий с плавающей запятой. Итак, посмотрим, что мы можем сделать с вычислением y x 320?

Вспомним, что мы используем двоичную арифметику и все числа в ПК также представлены в двоичном виде. Если вы берете двоичное число и сдвигаете его влево или вправо, это аналогично его умножению или делению на два.

Рисунок 5.3 поясняет это.

Рис. 5.3. *Битовый сдвиг для реализации умножения и деления.*

Поскольку операция сдвига выполняется примерно в 2-10 раз быстрее, чем умножение, то мы получим быстрые функции для рисования. Единственная сложность состоит в том, что число 320 — это не степень двух, но чтобы выйти из положения, мы применим маленькую хитрость. Представим выражение 320 x y, как 256 x, y + 64 x y. Листинг 5.5 показывает код для быстрого рисования точки.

Листинг 5.5. Программа быстрого рисования точки.

```
void Plot_Pixel_Fast ( int x, int y, unsigned char color )
{
// эта функция рисует точку несколько быстрее за счет замены
// операции умножения на сдвиг
// учитываем, что 320*y=256*y+64*y=y<<8+y<<6
video_buffer[((y<<8) +(y<<6)) + x ] = color;
} // конец функции
```

Эта функция работает примерно в два раза быстрее — вот что значит оптимизация. Позже мы научимся оптимизировать программы так, что парни из Microsoft перестанут нам верить.

Теперь, когда у нас есть функция рисования точки, надо дополнить нашу графическую библиотеку функцией рисования линии.

Рисование линий

Нас совершенно не интересует написание функций для рисования линий с применением сложностей типа алгоритма Брезинхема. Более того, в наших играх вообще не будет линий, проходящих под произвольным углом. Напротив, у нас будет множество вертикальных линий и, может быть, немного горизонтальных. На самом деле, большинство игр вообще обходится без рисования линий, так как в них используются битовые массивы.

Мы напишем две функции. Одна из них рисует горизонтальные линии слева направо, а другая — вертикальные сверху вниз. Рисунок 5.4 показывает, как они выглядят в видеобуфере.

Поскольку горизонтальные линии рисовать легче, то с них мы и начнем. Как видно из рисунка 5.4, горизонтальную линию можно получить, заполняя ряд пикселей в матрице 320x200. Чтобы это проделать, мы должны найти начальный адрес строки и заполнить ее значениями пикселей от начальной до конечной позиции. Для этого стоит использовать функцию rnemset. Это один из самых быстрых способов. Листинг 5,6 содержит код такой функции.

Листинг 5.6. Рисование горизонтальной линии.

```
void H_Line (int x1, int x2, int y, unsigned int color)
{
// функция рисует горизонтальную линию, используя memset()
// x2 должно быть больше x1
_fmemset ((char far *) (video_buffer + ((y << 8) + (y <<6)) + x1), color, x2 - x1 + 1);
} // конец функции
```

Следует кое-что запомнить:

- Мы используем функцию _fmemset, поскольку она, в отличие от memset, корректно работает с дальними (FAR) указателями;

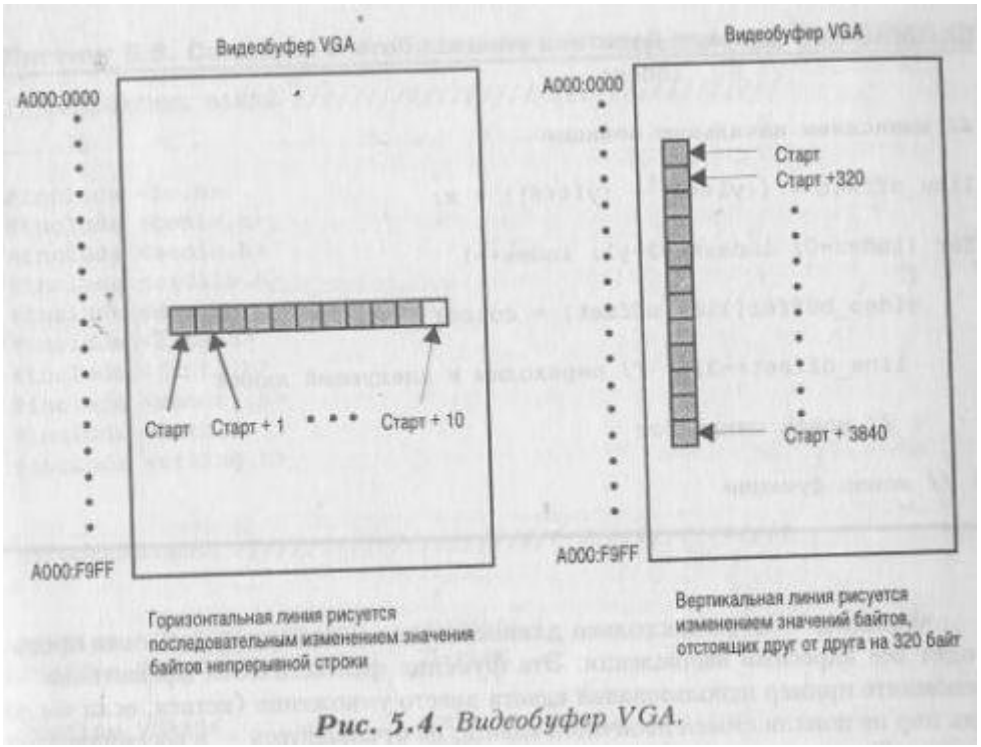


Рис. 5.4. Видеобуфер VGA.

- Мы вычисляем начальный адрес линии, и функция заполняет определенное количество байтов заданным значением цвета;
- Количество байтов равно длине строки, которую мы вычисляем как разность между правым и левым концами линии. При этом нужно прибавить единицу, чтобы не потерять последнюю точку.

К сожалению, мы не можем использовать семейство функций memset для рисования вертикальных линий, поскольку они работают только с непрерывными областями памяти. В случае вертикальных линий каждый следующий пиксель отстоит на 320 байт от предыдущего. Если вы находитесь в какой-то точке и хотите рисовать линию вниз, то вам надо прибавлять 320, а если линия рисует вверх, то надо вычитать 320 из текущего адреса. Таким образом, мы можем создать цикл, который увеличивает адрес на 320 и рисует вертикальную линию в виде пикселей сверху вниз. Листинг 5.7 демонстрирует код этой функции.

Листинг 5.7. Рисование вертикальной линии.

```
void V_Line(int yl,int y2,int x,unsigned int color)
{
//рисуем вертикальную линию (y2 больше yl)
unsigned int line offset,
                index;
// вычисляем начальную позицию
line_offset = ((y1<<8) + (y1<<6)) + x;
for (index=0; index<=y2-y1; index++)
{
video_buffer[line_offset] = color;
line_offset+=320;
// переходим к следующей линии
} // конец цикла for
} // конец функции
```

Функция V Line несколько длиннее H Line, поскольку она сама производит все адресные вычисления. Эта функция фантастически эффективна - вспомните пример использования сдвига вместо умножения (кстати, если вы до сих пор не поняли смысл двоичного сдвига, не отчаивайтесь - в восемнадцатой главе, «Техника оптимизации», мы это подробно изучим).

Прежде чем перейти к следующей теме, я хочу дать вам замечательную программу, которая создает новую палитру и выводит ее на экран, используя функции рисования вертикальных линий. В ней встречается уже известная функция Set_Mode() , описанная во второй главе, поэтому здесь я не включил ее исходный код. При желании вы можете взять его из второй главы. Более того, эта функция объявлена в программе как EXTERNAL, так что ее можно просто прилинковать. Листинг 5.8 содержит необходимый код программы Show_Palette.

Примечание

Я обнаружил некоторую проблему с чтением регистра палитры на VGA-картах. Похоже, что вы не всегда можете получить доступ к требуемому вам регистру. Это ошибка «железа», и она характерна для отдельных VGA-карт. Я решил эту проблему довольно примитивным способом: просто дважды читаю каждый регистр. Вроде, мне это помогло. В своих программах вы можете попробовать включать дополнительные проверки и использовать разные функции для чтения регистров палитры в случае обнаружения ошибки. Во всяком случае, время, потраченное на решение данной задачи с лихвой окупится в процессе выполнения программы.

Листинг 5.8. Создание и отображение цветовой палитры (PALDEMO.C).

```
// ВКЛЮЧАЕМЫЕ
ФАЙЛЫ ////////////////////////////////////////////////////////////////////
#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <bios.h>
#include <fcntl.h>
#include <memory.h>
#include <math.h>
#include <string.h>

// ОПРЕДЕЛЕНИЯ ////////////////////////////////////
#define ROM_CHARSET_SEG   0xF000
#define ROM_CHAR_SET_OFF  0xFA6E
#define VGA256             0x13
#define TEXT_MODE           0x03
#define PALETTE_MASK        O3c6
#define PALETTE_REGISTER_RD.O3c7 #define PALETTE_REGISTER_WR  0x3c8
#define PALETTE_DATA        0x3c9
#define SCREEN_WIDTH        (unsigned int)320
#define SCREEN_HEIGHT       (unsigned int)200
// СТРУКТУРЫ ДАННЫХ//////////////////////////////////////
// структура, сохраняющая RGB
typedef struct RGB_color_typ
{
unsigned char red;    // красный компонент 0-63
unsigned char green; // зеленый компонент 0-63
unsigned char blue;   // синий компонент 0-63
} RGB_Color, *RGB_color_ptr;
// ВНЕШНИЕ ФУНКЦИИ ////////////////////////////////
extern Set_Mode(int mode);
// ПРОТОТИПЫ //////////////////////////////////////
void Set_Palette Register(int index, RGB_color_ptr color);
void Get_Palette_Register(int index, RGB_color_ptr color);
void Create_Cool__Palette();
void V_Line(int y1,int y2,int x,unsigned int color);
// ГЛОБДЛЬНЫЕ ПЕРЕМЕННЫЕ //////////////////////////
// указатель на начало видеопамяти (для операций с байтами)
unsigned char far *video_buffer = (char far *)0xA0000000L;
// указатель на начало видеопамяти (для операций со словами)
unsigned int far *video_buffer_w= (int far *)0xA0000000L;
// ФУНКЦИИ ////////////////////////////////////////
void Set_Palette_Register (int index, RGB_color_ptr color)
{
// Эта функция устанавливает значение одного элемента таблицы
// цветов. Номер регистра задается переменной index, структура
// color содержит значения красной, зеленой и синей составляющих
```

```
// цвета
// указываем VGA карте, что мы будем обновлять содержимое
// регистра палитры
_outp(PALETTE_MASK,0xff);
// какой из регистров мы хотим обновить?
_outp(PALETTE_REGISTER_WR, index) ;
// теперь обновляем RGB
_outp(PALETTE_DATA,color->red);
_outp(PALETTE_DATA,color->green);
_outp(PALETTE_DATA,color->blue);
} // конец функции
//////////////////////////////////////////////////
void Get_Palette_Register(int index, RGB_color_ptr color)
{
// эта функция читает данные элемента таблицы цветов и помещает их
// в поля структуры color
// установить маску регистра палитры
_outp(PALETTE_MASK,0xff);
// сообщаем VGA, какой из регистров мы будем читать
_outp(PALETTE_REGISTER_RD, index);
// читаем данные
color->red   = _inp(PALETTE_DATA);
color->green = _inp(PALETTE_DATA);
color->blue = _inp(PALETTE_DATA);
} // конец функции            //////////////////////////////////////////////////
void Create_Cool_Palette(void) {
// эта функция создает палитру, содержащую по 64 оттенка серого,
// красного, зеленого и синего цветов
RGB_color color;
int index;
// проходим по элементам таблицы цветов и создаем 4 банка
// по 64 элемента
for (index=0; index < 64; index++)
{
// оттенки серого
color.red   = index;
color.green = index;
color.blue = index;
Set_Palette_Register(index, (RGB_color_ptr)&color);
// оттенки красного
color.red   = index;
color.green = 0;
color.blue = 0;
Set_Palette_Register(index+64, (RGB_color_ptr)&color) ;
// оттенки зеленого
color.red   = 0;
color.green = index;
color.blue = 0;
Set_Palette_Register(index+128, (RGB_color_ptr)&color);
// оттенки синего
color.red   = 0;
color.green = 0;
color.blue = index;
Set_Palette_Register(index+192, (RGB_color_ptr)&color);
} // конец цикла for
} // конец функции
//////////////////////////////////////////////////////
void V_Line(int y1,int y2,int x,unsigned int color)
{
// рисуем вертикальную линию y2 > yl
unsigned int line_offset, index;
// вычисляем начальную позицию
line_offset = ((y1<<8) + (y1<<6)} + x;
for (index=0; index<=y2-y1; index++)
{
video_buffer[line_offset] = color;
line_offset+==320; // переходим к следующей линии
```

```
} // конец цикла for
} // конец функции
// ОСНОВНАЯ ПРОГРАММА ////////////////////////////////
void main(void)
{
int index;
RGB_color color,color_1;
// установить режим 320x200x256
Set_Mode(VGA256) ;
// создать палитру цветов
Create_Cool_Palette();
// рисуем по одной вертикальной линии для каждого цвета
for (index=0; index<320; index++) V_Line(0,199,index,index);
// ждем реакции пользователя
while(!kbhit())
{
Get_Palette_Register(0,(RGB_color_ptr)&color 1) ;
Get_Palette_Register(0,(RGB_color_ptr)&color_l);
for (index=0; index<=254; index++)
{
Get_Palette_Register(index+l,(RGB_color_ptr)&color);
Get_Palette__Register(index+l, (RGB_color_ptr)&color) ;
Set Palette Register(index,(RGB color_ptr)&color) ;
} // конец цикла for
Set_Palette_Register(255,(RGB_color_ptr)&color_1);
} // конец цикла while
// переходим обратно в текстовый режим
Set_Mode(TEXT_MODE);
} // конец функции
```

Программа из Листинга 5.8 создает новую палитру, которая содержит 64 оттенка всех основных цветов, включая серый. Затем, она разделяет каждый цвет вертикальными линиями и после этого перемешивает их.

Вроде, хватит об этом. Теперь стоит поговорить о том, как целиком прочитать файл с образом. Начнем с формата PCX-файлов.

Графический формат PCX

В индустрии компьютерной графики существует так много стандартов, что само слово «стандарт» уже потеряло свой первоначальный смысл. Сегодня существует несколько наиболее известных стандартов: PCX, GIF, RGB, TGA, TIF и многие другие. Нам интересен формат PCX потому, что сегодня он является самым распространенным.

Файл в формате PCX представляет собой закодированное представление изображения. Кодирование необходимо для уменьшения размера файла, поскольку только один образ 320x200 пикселей уже займет 64K памяти. Рисованные объекты обладают большой цветовой избыточностью, и это обстоятельство используется для сжатия изображения.

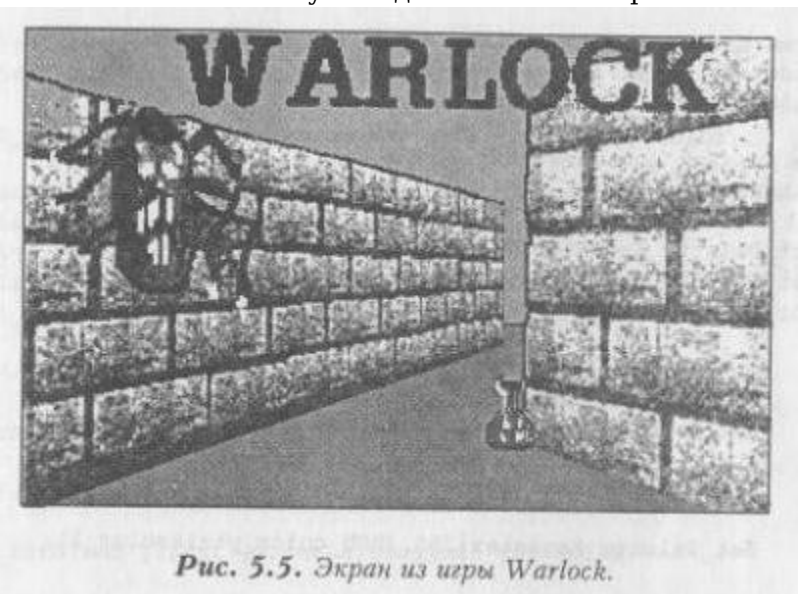


Рис. 5.5. Экран из игры Warlock.

Примечание

Отсканированные фотографии, как правило, содержат большое количество цветов, более ли менее

равномерно распределенных по всему изображению. Таким образом, фотографические изображения с трудом поддаются сжатию. Однако, поскольку большинство картинок в компьютерных играх выполняется вручную, мы можем не особенно ломать голову над проблемами, возникающими при работе с оцифрованными фотографиями**.**

Давайте посмотрим на рисунок 5.5 (файл CH19\WARINTR2.PCX на дискете). Это копия экрана из игры Warlock. Как вы можете заметить, там не слишком много цветов. Более того, на нем присутствует множество больших, одинаково окрашенных областей.

Как правило, в экранных изображениях используется лишь ограниченное число цветов. Так почему бы не подсчитать количество пикселей одинакового цвета и не сохранить их как целые группы, вместе с позицией и цветом. Это можно сделать. В общем, подобная технология, правда, в несколько усовершенствованном виде, и применена в PCX-формате. Для сжатия информации этот формат использует так называемое RLE-кодирование. При этом изображение обрабатывается построчно без учета расположения пикселей по вертикали.

Посмотрите на рисунок 5.6. Преобразование экранного файла в PCX-формат происходит следующим образом; просматривается очередная строка изображения, и если это возможно, сжимается. Сжатие выполняется в том случае, если найдена последовательность пикселей одинакового цвета. При этом сохраняется

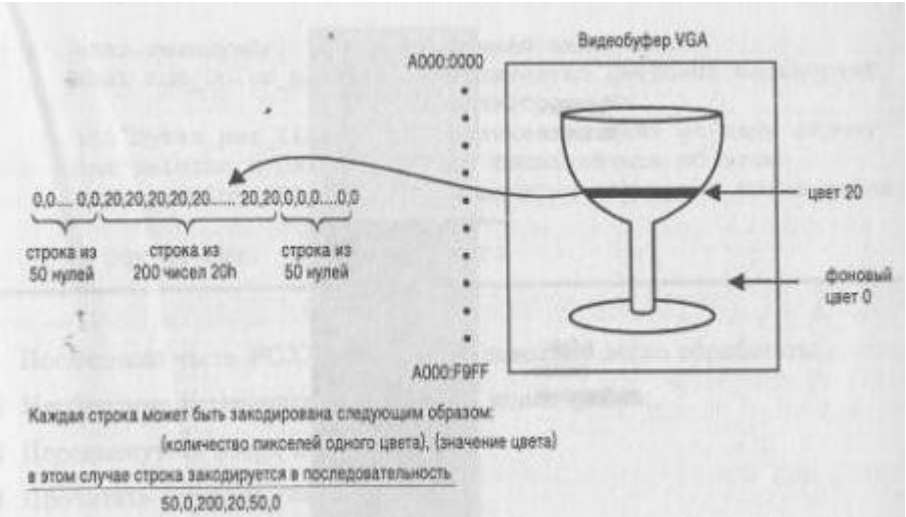


Рис. 5.6. *Схема сжатия методом RLE.*

количество повторений пикселей и значение их цвета. Повторяется это до тех пор, пока все изображение не окажется сжатым.

Такая методика неплохо работает для большинства изображений, но в некоторых случаях может не срабатывать, например, когда изображение имеет слишком мало одинаковых участков. В этом случае размер файла, напротив, увеличится, так как для записи одиночных пикселей требуется не один, а два байта.

Файл формата PCX состоит из трех секций:

- Первая секция PCX-файла длиной 128 байт содержит различную служебную информацию;
- Вторая секция — это данные сжатого образа, которые могут оказаться любой длины;
- Третья секция размером в 768 байт содержит цветовую палитру, если она есть. В нашем случае она будет присутствовать, поскольку мы используем 256-цветный режим 13h. Эти 768 байт хранят значения RGB от 0 до 255.

Суммируя вышесказанное, можно нарисовать структуру PCX-файла (рис. 5.7).

Получение информации из заголовка несложно: достаточно прочитать первые 128 байт и отформатировать их в соответствии со структурой, представленной в Листинге 5-9.

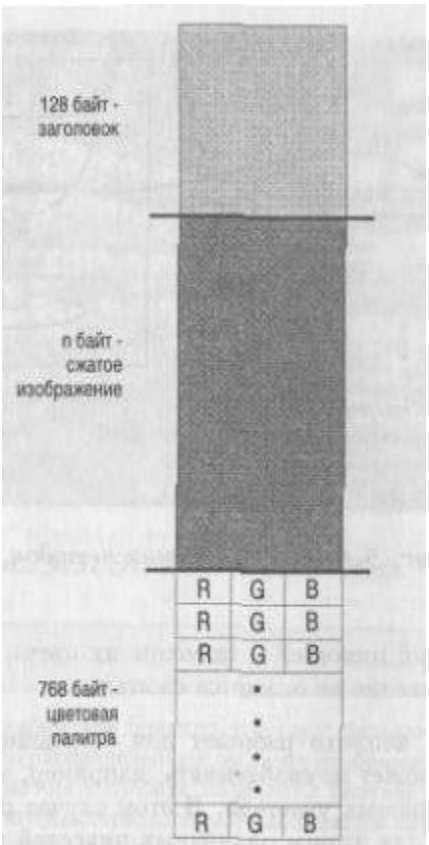


Рис. 5.7. *Формат PCX.*

Листинг 5.9. Структура заголовка PCX-файла.

```
typedef struct pcx_header_typ
{
char manufacturer; // всегда 10
char version;       // 0 - версия 2.5 Paintbrush
                    // 2 - версия 2.8 с палитрой
                    // 3 - версия 2.8 без палитры
                    // 5 - версия 3.0 или старше
char encoding;      // всегда 1 - RLE кодирование
char bits_per_pixel;// количество бит на пиксель
                    // для нашего случая – 8
int x,y; // координаты верхнего левого угла изображения
int width,height; // размеры изображения
int horz_res;      // количество пикселей по горизонтали
int vert_res;      // количество пикселей по вертикали
char ega_palette[48]; // EGA-палитра. Ее можно игнорировать,
char reserved;         // ничего значимого
char num_color_planes; // количество цветовых плоскостей
                       //в изображении
int bytes_per_line;    // количество байт на одну строку
int palette_type;      // не беспокойтесь об этом
char padding[58];      // ссылка на палитру в конце файла
} pcx_header, *pcx_header_ptr;
```

Последнюю часть PCX-файла также довольно легко обработать:

- Необходимо установить указатель на конец файла;
- Передвинуться вверх на 768 байт;
- Прочитать 768 байт как палитру.

Конечно, я упустил кое-какие детали обработки PCX-файла, но сделал это лишь для того, чтобы лучше передать смысл производимых действий. Сейчас же нас больше должен занимать способ декодирования средней части, где находится само изображение. Именно отсюда начинаются сложности, поскольку процедура декомпрессии не очень проста и очевидна.

- Если код прочитанного байта принадлежит множеству 192...255, то мы вычитаем из него 192 и используем полученный результат, как количество повторений следующего байта;
- Если код прочитанного байта лежит в диапазоне от 0 до 191, то мы используем его как байт данных, то есть помещаем его в битовую карту без изменений.

Если вы достаточно внимательны, то можете спросить: «А как же быть с пикселями, имеющими значения от 192 до 255? Интерпретируются ли они как RLE-цепочки?» Да, и гениальное решение этого вопроса состоит в том, что такие значения кодируются не одним, а двумя байтами. Например, если требуется поместить в файл значение 200, то сначала нужно записать число 193 (192-1) как количество повторений, а потом — 200. Посмотрим на рисунок 5.8, чтобы увидеть пример декомпрессии.

Теперь настало время написать программу, реализующую чтение файл формата PCX. Она [illegible] возможность убедиться в этом самостоятельно.

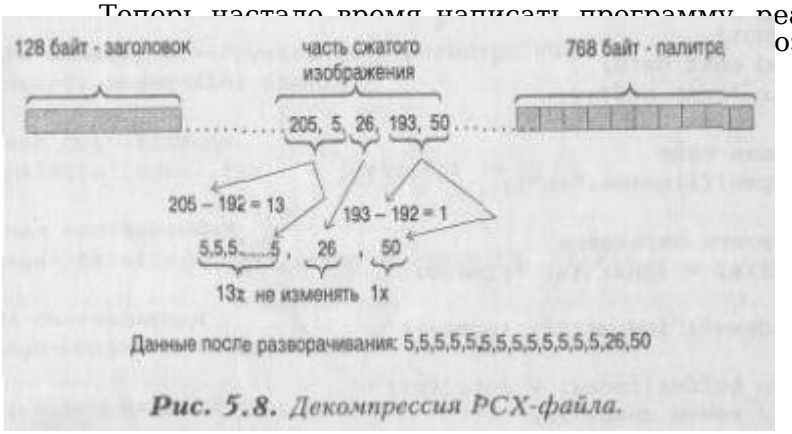


***Рис. 5.8.** Декомпрессия PCX-файла.*

Листинг 5.10. Программа чтения файла формата PCX.

```
// размеры экрана
#define SCREEN_WIDTH 320
#define SCREEN_HEIGHT 200
// структура для хранения данных PCX файла
```

```
typedef struct pcx_picture_typ
{
pcx_header header; // заголовок файла (длина 128 байт)
RGB_color palette[256]; // палитра
char far *buffer; // буфер для размещения изображения
// после декомпрессии
} pcx_picture, *pcx_picture_ptr;
void PCX Load(char *filename,
pcx_picture_ptr image,int enable_palette)
{
// функция загружает данные из PCX-файла в структуру pcx picture
// после декомпрессии байты изображения помещаются в буфер.
// Отдельные элементы изображения выделяются позднее. Также
// загружается палитра и заголовок
FILE *fp;
int num_bytes,index;
long count;
unsigned char data;
char far *temp_buffer;
// открыть файл
fp = fopen(filename,"rb");
// загрузить заголовок
temp_buffer = (char far*)image;
for (index=0; index<l28; index++)
{
temp_buffer[index] = getc(fp);
} // конец цикла for
// загрузить данные и декодировать их в буфере
count=0;
while (count<=SCREEN_WIDTH * SCREEN_HEIGHT}
{
// получить первую часть данных
data = getc(fp);
//это  RLE?
if (data>=192 &&data<=255)
{
// подсчитываем, сколько байт сжато
num_bytes = data-192;
//читаем байт цвета
data = getc(fp);
// повторяем байты в буфере num_bytes раз
while(num_bytes-->0)
{
image->buffer[count++] = data;
} // конец цикла while
} // конец оператора if
else
{
// помещаем данные в следующую позицию буфера
image->buffer[count++] = data;
} // конец оператора else
} // конец чтения байт изображения
// перейти в позицию, не доходя 768 байт от конца файла
fseek(fp,-768L,SEEK_END);
// читаем палитру и загружаем ее в регистры VGA
for (index=0; index<256; index++)
{
// красная составляющая
image->palette[index].red   = (getc(fp) >> 2);
// зеленая составляющая
image->palette[index].green = (getc(fp) >> 2);
// синяя составляющая
image->palette[index].blue = (getc(fp) >> 2);
} // конец цикла for

fclose(fp);
// если флаг enable_palette установлен, меняем палитру
```

```
// на загруженную из файла
if (enable_palette)
{
for (index=0; index<256; index++)
{
Set_Palette_Register(index,
(RGB_color_ptr)&image->palette[index]);
} // конец цикла for
} // конец установки новой палитры
} // конец функции
```

Функция PCX_Load() — это сердце всей программы. Она загружает PCX-файл, декодирует его в буфере и загружает палитру. Каждый PCX-файл имеет свою собственную палитру в конце файла и я думаю, что вы сами можете добавить возможность загрузки повой палитры в таблицу соответствия цветов.

Функция выполняет именно те действия, которые-мы с вами уже обсуждали и ничего больше:

- Открывает PCX-файл;
- Читает заголовок;
- Загружает PCX-файл и декомпрессирует все 64000 пикселей;
- Загружает цветовую палитру.

В общем, все это несложно. А вот что делать с картинками, которые больше, чем целый экран? Ответ прост: можно декодировать только маленький кусочек, скажем 24 на 24 пикселя.

Я создал для вас заготовку CHARPLATE.PCX, которую вы найдете на прилагаемом диске. Если вы посмотрите на него, то увидите множество маленьких белых квадратов. Вы можете использовать этот шаблон для рисования ваших игровых персонажей в этих квадратиках.

Кстати, впоследствии мы научимся извлекать битовые образы из больших PCX-файлов и использовать их в качестве персонажей игры.

Возникает вопрос: «Как редактировать PCX-файлы в режиме 320x200x 256?» Для этого можно воспользоваться такими условно-бесплатными программами как VGA-Paint или Pro-Paint. Тем не менее, я надеюсь, что самые расторопные читатели уже давно пользуются копией Electronic Art's Deluxe Paint & Animation. Это одна из самых классных программ для рисования на ПК. Она корректно работает с режимом 320x200x256 и имеет множество полезных функций для преобразования и анимации изображения.

Побитовое копирование изображения (бит-блиттинг)

Термин бит-блиттинг (bit blitting) — означает процесс перемещения группы битов (образа) из одного места экрана в другое. В играх для ПК нас интересует перемещение образа из области хранения вне экрана в область видеобуфера. Давайте посмотрим на рисунок 5.9, чтобы уяснить сущность этой операции.

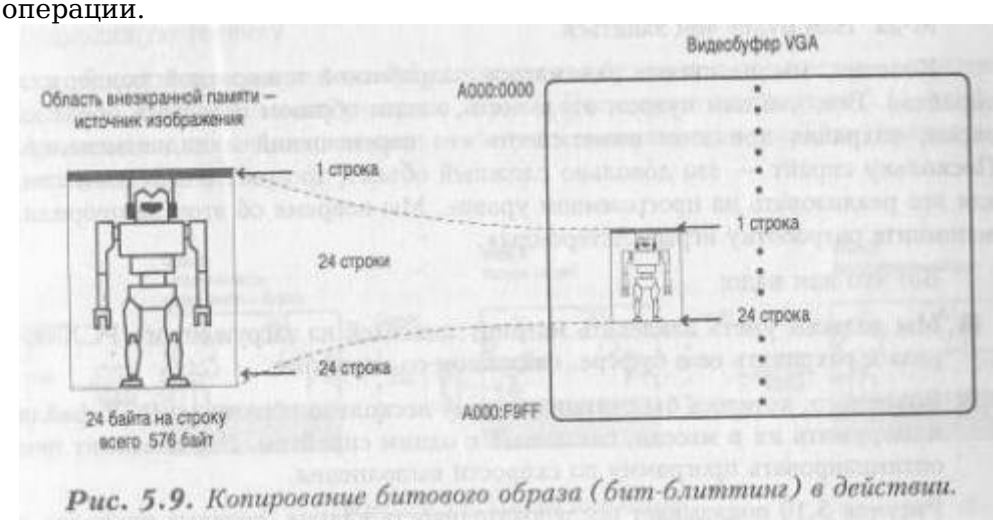


Рис. 5.9. Копирование битового образа (бит-блиттинг) в действии.

Как вы можете видеть, матрица пикселей обычно копируется один к одному из исходной области хранения в область буфера экрана. Кроме того, еще встречаются ситуации, когда что-то из видеобуфера копируется в память для дальнейшего использования.

Чтобы понять суть перемещения, нам следовало бы написать несколько функций, которые бы брали битовую карту из PCX-файла и перемещали ее на экран. Но я хочу проявить некоторую «авторскую вольность» и поговорить о спрайтах и их анимации.

Спрайты

Вы можете спросить: «Что такое спрайт?». Знаете, есть такой газированный напиток... Снова шучу. На самом деле спрайты - это такие маленькие объектики, которые находятся на игровом поле и могут двигаться. Этот термин

прижился с легкой руки программистов фирмы Atari и Apple в середине, 70-х годов. Теперь поговорим о спрайтах и их анимации. В будущем мы еще вернемся к этой теме в седьмой главе, «Продвинутая битовая графика и специальные эффекты. Именно с этой мыслью я создал несколько небольших спрайтов, которые мы будем использовать в дальнейшем.

Спрайты - это персонажи в играх для ПК, которые могут без труда перемещаться по экрану, изменять цвет и размер. Все это звучит как мечта программиста. Но надо помнить, что в IBM-совместимых компьютерах нет спрайтов! Во нормальных компьютерах существует аппаратная поддержка спрайтов. Такие машины как Atari, Amiga, Commodore и последние модели Apple имеют эту возможность, а вот ПК - нет. Поэтому мы вынуждены делать это самостоятельно.

М-да. Нам будет чем заняться.

Конечно, мы не станем заниматься разработкой аппаратной поддержки спрайтов. Все, что нам нужно, это понять, каким образом помещать образ на экран, сохраняя при этом возможность его перемещений и видоизменений. Поскольку спрайт — это довольно сложный объект, то стоит подумать о том, как это реализовать на программном уровне. Мы вовремя об этом заговорили:. вспомните разработку игры «Астероиды».

Вот что нам надо:

- Мы должны уметь извлекать матрицу пикселей из загруженного PCX-образа и сохранять ее в буфере, связанном со спрайтом;
- Более того, хотелось бы считывать сразу несколько образов из PCX-файла и загружать их в массив, связанный с одним спрайтом. Это позволит нам оптимизировать программу по скорости выполнения.

Рисунок 5.10 показывает последовательность кадров, которые приводят в движение ковбоя. Мы воспользуемся ею позже.

Рис. 5.10. Идущий ковбой.

После того как мы загрузим данные из PCX-файла, нам необходимо иметь возможность показывать спрайт в любой позиции на экране. Делать это нужно осторожно, поскольку запись пикселей в видеобуфер разрушает то, что было на их месте. Поэтому, мы должны уметь сохранять ту часть изображения, которая окажется закрыта спрайтом, чтобы в дальнейшем иметь возможность восстановить первоначальный вид экрана.

Давайте на этом Месте остановимся и поговорим чуть-чуть об анимации. В играх для ПК применяется два способа обновления экрана:

- Мы можем перерисовывать весь экран целиком, как это сделано; в игре Wolfenstein 3D;
- Можно перерисовывать лишь участки экрана.

Какой из способов лучше выбрать, зависит от типа игры. Если мы перерисовываем весь экран, то это нужно делать по возможности быстро, поскольку 64000 пикселей - все же довольно много. Если мы перерисовываем только участки экрана, то желательно быть уверенным, что фон после прохождения спрайта не изменится. Поскольку все игры для ПК отличаются друг от друга, то для решения конкретных специфических задач всегда надо выбирать наиболее подходящую технику.

Давайте рассмотрим способ, которым мы будем пользоваться в настоящей главе - это перерисовка участков экрана. Посмотрите на рисунок 5.11, чтобы представить последовательность событий, позволяющих спрайту правильно перемещаться по экрану.

Рис. 5.11. Фазы анимации.

Теперь, когда мы знаем что делать, надо попробовать это реализовать. Для начала создадим структуру данных спрайта. Листинг 5.11 содержит необходимый для этого код.

Листинг 5.11. Структура спрайта с полями для анимации.

```
typedef struct sprite_typ
{
int x,y;            // текущая позиция спрайта
int x_old, y_old;    // предыдущая позиция спрайта
```

```
int width,height;   // размеры спрайта
int anim_clock;     // время анимации
int anim_speed;     // скорость анимации
int motion_speed;   // скорость движения
int motion_clock;   // время движения
char far *frames [MAX_SPRITE_FRAMES] ; // массив указателей
//на кадры
int curr_frame;                       // отображаемый кадр
int num_frames;                       // общее число кадров
int state;                            // статус спрайта
char far *background;                 // фон под спрайтом
}sprite, *sprite_ptr;
```

Структура спрайта имеет поля для сохранения позиции и размеров образа и несколько других элементов. Сейчас мы уже готовы написать функцию для работы со спрайтами.

Прежде всего мы должны извлечь битовую карту из PCX-файла и поместить ее в массив, хранящий образы спрайта. Если вы помните, я создал файл в формате PCX (CHARPLATE.PCX), в который вы можете дорисовать свои картинки и героев. Функция, извлекающая битовые карты из PCX-образа подразумевает, что вы создали свои образы с помощью этого файла. Программа из Листинга 5.12 позволяет перемещать спрайт, который вы хотите изменить, в указанные координаты.

Листинг 5.12. Функция извлечения спрайта из загруженного PCXфайла.

```
void PCX_Grap_Bitmap(pcx_picture_ptr image, sprite_ptr sprite, int sprite_franie, int
grab_x, int grab_y)
{
// функция выделяет одно изображение из буфера в который
// загружен PCX-файл
// функция исходит из предположения, что в действительности массив
// пикселей размером 320x200 разделен на отдельные изображения
// размером 24x24 пикселя
int x_off,y_off, x,y, index;
char far *sprite_data;
// вначале выделяем память для хранения спрайта в структуре спрайта
sprite->frames[sprite_frame] = (char far *)malloc(SPRITE_WIDTH * SPRITE_HEIGHT);
// создаем альтернативный указатель на эту область памяти
// для ускорения доступа
sprite_data = sprite->frames[sprite_frame];
// теперь перемещаем битовый образ спрайта из области PCX-файла
// в выделенную память
// мы должны выбрать, какой именно спрайт мы копируем
// помните, что в действительности файл представляет собой массив.
// 12x8 элементов, каждый из которых имеет размер 24x24 пикселя.
// Индекс (0,0) соответствует верхнему левому углу спрайта,
// (11,7) - нижнему правому
x_off = 25 * grab_x + 1;
y_off = 25 * grab_y + 1;
// вычисляем начальный адрес
y_off =y_off * 320;
for (y=0; y<SPRITE_HEIGHT; y++)
{
for (x=0; x<SPRITE_WIDTH; X++)
{
// получить очередной байт текущей строки и поместить его
//в следующую позицию буфера спрайта
sprite_data[y*24 + x] = image->buffer[y_off + x_off + x];
} // конец копирования строки
// перейти к следующей строке
y_off+=320;
} // конец копирования
// инкрементировать счетчик кадров
sprite->num_frames++;
} // конец функции
```

Эта функция по указателю на спрайт определяет его расположение в загруженном файле. Далее она выделяет память для хранения образа и инициализирует структуру данных (я решил делать спрайты размером 24 на 24 пикселя, но вам ничто не мешает изготавливать любые другие спрайты). Теперь, когда у нас подготовлены образы, следующим шагом будет их отображение на экране монитора. Для этого нам надо:

- Вычислить начальную позицию спрайта согласно его координатам (x,y);
- Преобразовать полученные координаты в адрес видеобуфера;
- Переместить байты изображения в видеобуфер.

Для рисования спрайта мы должны выполнить все операции с текущим кадром анимации. Код в Листинге 5.13 делает все перечисленное.

Листинг 5.13. Функция рисования спрайта.

```
void DrawSprite (sprite_ptr sprite)
{
// функция, рисующая спрайт на экране строка за строкой,
// очень быстро. Вместо умножения используется сдвиг
char far *work_sprite;
int work_offset=0,offset,x,y;
unsigned char data;
// создаем альтернативный указатель на спрайт для ускорения доступа
work_sprite = sprite->frames[sprite->curr frame];
// вычислить смещение спрайта в видеобуфере
offset = (sprite->y << 8) + (sprite->y << 6) + sprite->x;
for (y=0; y<SPRITE_HEIGHT; y++)
{
for (x=0; x<SPRITE_WIDTH; x++)
{
// Проверка на "прозрачный" пиксель (с кодом 0). Если пиксель
// "непрозрачный" - выводим его на экран.
if ((data=work_sprite[work_offset+x])) video_buffer[offset+xj = data;
} // конец вывода строки
// перейти к следующему ряду пикселей в видеобуфере
//в буфере спрайта
offset      += SCREEN_WIDTH;
work_offset += SPRITE_WIDTH;
} // коней вывода спрайта
} // конец функции
```

Эта функция работает примерно так же, как и Plot_Pixel_Fast из Листинга 5.5. Сначала вычисляется стартовый адрес расположения спрайта, а затем все его байты строка за строкой переписываются в видеобуфер.

Следует учесть, что здесь потребуется некоторая оптимизация. Почему бы не использовать функцию memcpy, чтобы копировать всю строку разом (а всего 24 строки)? Однако этого сделать [illegible] применить технику, использующую [illegible] Взгляните на рисунок 5.12.

Рис. 5.12. «Прозрачность».

Спрайт слева выведен вместе с фоном, а при рисовании правого спрайта использовалась техника «прозрачных» пикселей. При этом пиксели с цветом фона (черный, имеющий код 0) пропускались, а прорисовывались только данные непосредственного изображения. Это и создает эффект «прозрачности» фона.

Следующая функция, которую я собираюсь вам предложить, будет сохранять фон перед тем, как мы выведем спрайт на экран. Помните, когда мы что-то записываем в видеобуфер, данные или образ, находящийся там, обязательно теряются. Поэтому мы и должны сохранять фон под спрайтом прежде чем поместим его в видеобуфер, чтобы позже восстановить прежний вид экрана. Код в Листинге 5.14 именно это и делает.

Листинг 5.14. Сохранение фона под спрайтом.

```
void Behind_Sprite(sprite_ptr sprite)
{ // функция сохраняет область видеопамяти, в которую будет
// выводиться спрайт
char far *work_back;
in work_offset=0,offset,y;
// создаем альтернативный указатель для ускорения доступа
work_back = sprite->background;
```

```
// вычисляем смещение в видеобуфере
offset = (sprite->y << 8) + (sprite->y << 6) + sprite->x;
```

```
for (y=0; y<SPRITE_HEIGHT; y++)
{
// копируем строку видеопамяти в буфер
_fmemcpy((char far *)&work_back[work_offset], (char far *)&video_buffer[offset],
SPRITE_WIDTH);
// переходим к следующей строке
offset    += SCREEN_WIDTH;
work_offset += SPRITE_WIDTH;
} // конец цикла for
} // конец функции
```

Функция Behind_Sprite считывает матрицу размером 24x24, где должен находится спрайт. Данные после считывания находятся в поле background структуры спрайта. Это поле является указателем на область памяти, где находится спрайт.

Собственно, это все, что нам нужно для создания и перемещения маленьких образов по экрану. Для анимации мы должны изменять поле curr_frame в структуре спрайта, перед тем, как его начать рисовать. Мы обсудим процесс анимации в этой главе, но, я думаю, вы и сами догадываетесь, как это сделать: надо стереть спрайт, передвинуть его, снова нарисовать и т. д. Вот и все.

Отображение текста

Одна из самых последних тем, понимание которой очень важно для создания видеоигр, это печать текста. Если вы пропустите этот раздел, не прочитав ни слова, то очень ошибетесь, потому как не существует игр, в которых не встречалось бы никаких надписей.

Поскольку наши игры будут работать в графическом режиме, то единственный способ напечатать текст на экране - это выводить его по точкам. Мы можем нарисовать все 128 символов и загружать их из PCX-файла, но боюсь, что это похоронит все наши дальнейшие планы. Я бы лучше использовал битовые образы внутреннего набора символов, скрытые в памяти ПК. Если бы мы знали их местоположение, то смогли бы их использовать для вывода символов и надписей любого цвета и размера. Не верите? Ладно, я знаю, как это сделать, и сейчас научу вас.

Данные для таблицы символов ROM размером 8x8 начинаются с адреса F000:FA6E. Символы располагаются в ASCII-последоватедьности, то есть описание символа A находится на 65-м месте и т. д. Каждый символ состоит из восьми байт, где каждый байт представляет собой одну строку битовой карты символа. Мы можем умножить 65 на 8, чтобы найти смещение первого байта данных в наборе символов. Это смещение затем складывается с базовым адресом FOOO:FA6E, и следующие восемь байт представляют каждую из строк символа. Давайте посмотрим на рисунок 5.13 для большей ясности.

Единственная проблема режиме в 13h, которая у нас возникает, заключается в том, что для каждого пикселя требуется не один бит, а целый байт, в то время как каждая строка любого символа описывается байтом! И ничего с этим не сделать. Мы можем только окрашивать символы определенным цветом.

Алгоритм выглядит примерно так:

- Получить следующую строку данных символа (следующий байт);
- Нарисовать по горизонтали восемь точек, где пиксели с 0 по 7 установлены либо в 1, либо в 0, в зависимости от значения битов 0 - 7 в байте данных;

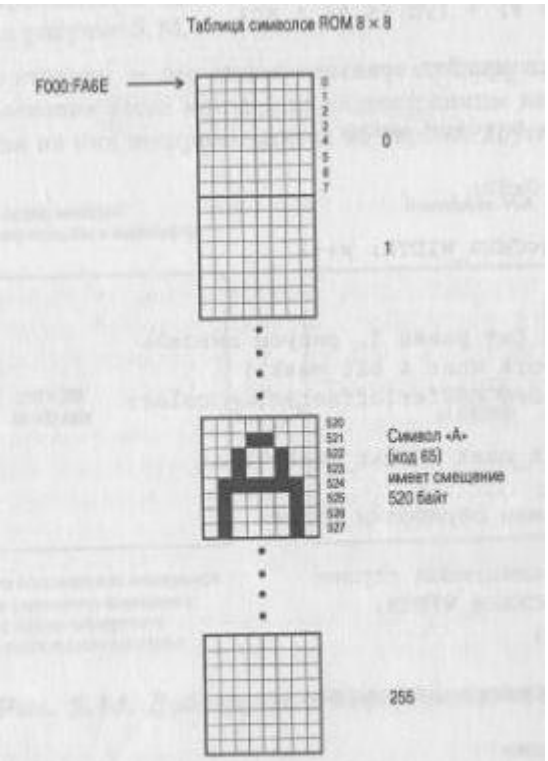


Рис. 5.13. *Таблица символов ROM 8×8.*

- Проделать то же самое для каждой строки данных символа (байта).

Листинг 5.15 содержит код для изображения символа любого цвета в любой позиции на экране.

Листинг 5.15. Отображение символа.

```
void Blit_Char(int xc,inf yc,char c,int color)
{
// Функция использует описание символов 8x8 байт, хранящееся в ПЗУ
// для изображения символа на экране. Хитрость заключена в том, что
// в этом описании каждая экранная точка представлена одним битом.
int offset,x,y;
unsigned char data;
char far *work_char;
unsigned char bit_mask = 0x80;
// вычисляем начальное смещение символа в ROM
work_char= rom_char_set + c * CHAR_HEIGHT;
// вычисляем смещение символа в видеобуфере
offset = (yc << 8) + (yc << 6) + xc;
for (y=0; y<CHAR_HEIGHT; y++)
{
// сбросить битовую маску
bit_mask = 0x80;
for (x=0; x<CHAR_WIDTH; x++)
{
// если бит равен 1, рисуем пиксель
if ((*work_char & bit_niask))
video_buffer[offset+x] = color;
bit_mask = (bit_mask>>l);
} // конец обработки строки
// переходим к следующей строке
offset += SCREEN_WIDTH;
work_char++;
} // конец рисования символа
} // конец функции
```

Теперь и печать текстов не представляет труда. Нужно только выдерживать пространство между символами и печатать их из заданной строки один за другим, пока не встретится конец строки (NULL). В демонстрационной программе в конце этой главы мы используем подобную функцию.

Теперь давайте поговорим о методах анимации и рендеринга, которые применяются технике программирования игр.

Дублирующее буферизирование

Дублирующее буферизирование — это метод, позволяющий избежать мерцания, которое может возникать при перемещении объектов. Когда функции, которые мы пишем, рисуют спрайты па экране, то они это делают, не учитывая статус состояния VGA-карты, то есть в них отсутствует синхронизация с дисплеем. Существует два способа уменьшить возможное мигание.

Первый метод называется дублирующим буферизированием. При этом экранное изображение формируется в памяти, а затем копируется в видеобуфер. Это позволяет минимизировать перемещение маленьких участков в видеопамяти, поскольку сразу перемещается один блок. Пример дублирующего буферизирования изображен на рисунке 5.14.

Переключение страниц - это видоизмененное дублирующее буферизирование. При использовании этого метода две видеостраницы находятся во взаимосвязи. Когда одна из них воспроизводится на экране, другая перерисовывается. Новая страница затем воспроизводится переключением указателя на нее. Обратите внимание на рисунок 5.15 для осмысления этого метода.

Оба этих метода имеют общий недостаток - уменьшение быстродействия в два раза, но при этом отсутствует мигание и достигается очень высокое качество анимации. Мы обратимся к этому методу еще раз в седьмой главе, «Продвинутая битовая графика и специальные эффекты». Сейчас поговорим об уменьшении мерцания.

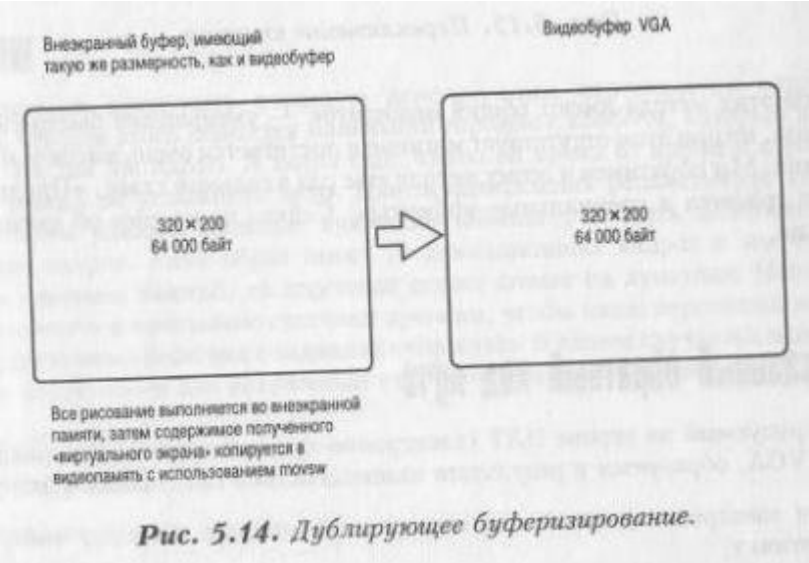


Рис. 5.14. *Дублирующее буферизирование.*

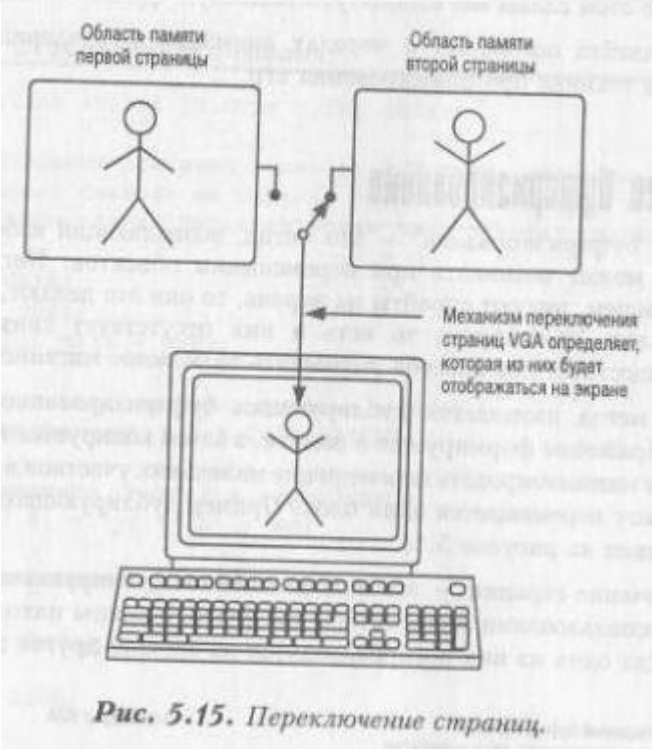


Рис. 5.15. *Переключение страниц.*

Вертикальный обратный ход луча

Образ, рисуемый на экране ЭЛТ (электронно-лучевой трубки) и управляемый картой VGA, образуется в результате взаимодействия следующих факторов:

- Луч электронов движется, по экрану слева направо и сверху вниз, рисуя картинку;
- Когда он достигает нижней границы, он вновь возвращается вверх и все начинается сначала.

Рисунок 5.16 показывает это.

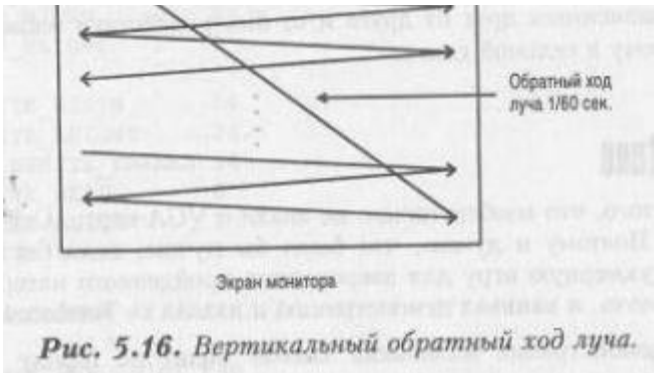


Рис. 5.16. *Вертикальный обратный ход луча.*

- Чтобы вернуться в исходную позицию, лучу требуется примерно 1/60 секунды. Это идеальное время для обновления видеобуфера. В течение этого периода видеобуфер недоступен VGA-карте. Таким образом, 1/60 секунды — это аппаратно-зависимый параметр.

В седьмой главе, «Продвинутая битовая графика и специальные эффекты», мы узнаем, как «синхронизировать» наши игры с этим сигналом и создать чистые, свободные от мерцания изображения.

Тайминг

Я попытался придумать хорошую демонстрацию перемещения спрайтов и решил, что для этого подойдет маленький городок с ковбоем, который ходит по улице. Не так уж плохо. Я хотел еще, чтобы он время от времени стрелял, но позже решил не усложнять дело. Для осуществления реалистичной анимации мы должны уделять большое внимание таймингу, то есть задержке между выводом кадров. Если образ имеет 10 анимационных кадров и мы будем их менять слишком быстро, то персонаж станет похож на лунатика. Поэтому мы должны иметь в программе счетчики времени, чтобы наши персонажи выполняли определенные действия с заданной скоростью. В нашем случае мы используем четыре переменные для сохранения счетчиков движения и анимации:

- anim_clock
- anim_speed
- motion_clock
- motion_speed

Переменные скорости - это константы, а переменные времени обновляются при каждом проходе через главный цикл. Когда переменная времени оказывается больше переменной скорости, то мы осуществляем какое-либо действие: перемещаем спрайт или сменяем кадр. Переменные времени дри этом обнуляются. Это позволяет нам сделать перемещение спрайта по экрану и смену фаз движения независимым друг от друга и от быстродействия машины. Мы еще вернемся к этому в седьмой главе.

Игра Tombstone

Мы начали с того, что вообще ничего не знали о VGA-карте. Сейчас мы знаем очень много. Поэтому я думаю, что было бы лучше, если бы вы написали маленькую двухмерную игру для закрепления пройденного материала. Чтобы вам в этом помочь, я написал демонстрацию и назвал ее Tombstone.

В этой демонстрации маленький ковбой ходит по городу с различной скоростью. У вас есть все инструменты, чтобы «дать» ему пистолет и "научить" стрелять. В PCX-файле на дискете вы найдете для этого все необходимые картинки. Вся программа за исключением функции Set_Mode() дана в Листинге 5.16. Прилинкуйте Set_Mode(), когда будете создавать исполняемый файл.

Листинг 5.16. Tombstone (TOMB.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ///////////////////////////////////////////
#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <bios.h>
#include <fcntl.h>
#include <memory.h>
#include <math.h>
#include <string.h>
// ОПРЕДЕЛЕНИЯ /////////////////////////////////////////////////
#define ROM_CHAR_SET_SEG 0xF000 // сегмент описания символов в ПЗУ
#define ROM_CHAR_SET_OFF 0xFA6E // смещение, соответствующее
// описанию первого символа
#define VGA256             0x13
#define TEXT_MODE          0x03
#define PALETTE_MASK          0x3C6
#define PALETTE_REGISTER_RD 0x3C7
#define PALETTE_REGISTER_WR 0x3C8
#define PALETTE_DATA         0x3C9
#define SCREEN_WIDTH      (unsigned int)320
#define SCREEN_HEIGHT     (unsigned int)200
#define CHAR_WIDTH        8
#define CHAR_HEIGHT       8
#define SPRITE_MIDTH      24
#define SPRITE_HEIGHT     24
#define MAX_SPRITE_FRAMES 16
#define SPRITE_DEAD       0
```

```
#define SPRITE_ALIVE      1
#define SPRITE_DYING      2
// СТРУКТУРЫ ДАННЫХ ////////////////////////////////////
typedef struct RGB_color_typ
{
unsigned char red;   // красная составляющая цвета (0-63)
unsigned char green; // зеленая составляющая цвета (0-63)
unsigned char blue; // синяя составляющая цвета (0-63)
} RGB_color, *RGB_color_ptr;

typedef struct pcx_header_typ
{ char manufacturer;
char version;
char encoding;
char bits_per_pixel;
int x,y;
int width,height;
int horz_res;
int vert_res;
char ega_palette[46];
char reserved;
char num_color_planes;
int bytes_per_line;
int palette_type;
char padding[58];
} pcx_header, *pcx_header_ptr;
typedef struct pcx_picture_typ
{
pcx_header header;
RGB_color palette[256];
char far *buffer;
} pcx_picture, *pcx_picture_ptr;
typedef struct sprite_typ
{
int x,y;            // текущая позиция спрайта
int x_old,y_old;    // предыдущая позиция спрайта
int width,height;   // размеры спрайта
int anim_clock;     // время анимации
int anim_speed;     // скорость анимации
int motion_speed;   // скорость движения
int motion_clock;   // время движения
char far *frames[MAX_SPRITE__FRAMES]; // массив указателей
// на образы
int curr_frame;                      // отображаемый фрейм
int num_frames;                      // общее число фреймов
int state;                           // статус спрайта
char far *background;               // фон под спрайтом
}sprite, *sprite_ptr;
// ВНЕШНИЕ ФУНКЦИИ ////////////////////////////////
extern Set_Mode(int mode);
// ПРОТОТИПЫ //////////////////////////////////////
void Set_Palette_Register(int index, RGB_color_ptr color);
void Plot_Pixel_Fast(int x,int y,unsigned char color);
void PCX_Init(pcx_picture *image);
void PCX_Delete(pcx_picture *image);
void PCX_Load(char *filename, pcx_picture_ptr image, int enable_palette) ;
void PCX_Show_Buffer(pcx_picture_ptr image);
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ //////////////////////////////
unsigned char far *video_buffer   = (char far *) 0xA0000000L;
unsigned int  far *video_buffer_w = (int  far *)0xA0000000L;
unsigned char far *rom_char_set   = (char far *)0xF000FA6EL;
// ФУНКЦИИ ////////////////////////////////////////////
void Blit_Char(int xc,int yc,char c,int color) {
// эта функция отображает символ на экране
// используя описание
// символов размером 8x8 точек, хранящееся в ПЗУ
int offset,x,y;
```

```
unsigned char data;
char far *work_char;
unsigned char bit_mask = 0x80;
// вычисляем смещение описания символа в ПЗУ
work_char = rom_charset + c * CHAR_HEIGHT;
// вычисляем смещение символа в видеобуфере
offset = (yc << 8} + (yc << 6) + xc;
for (y=0; y<CHAR_HEIGHT; y++)
{
// сбросить битовую маску
bit_mask = 0x80;
for (x=0; x<CHAR_WIDTH; x++)
{ // если бит равен 1, рисуем пиксель
if ((*work_char & bit_mask))
video_buffer[offset+x] = color;
// сдвигаем маску
bit_mask = (bit_mask>>1);
} // конец отрисовки строки
// переходим к следующей строке
offset      += SCREEN_WIDTH;
work_char++;
} // конец рисования
} // конец функции
////////////////////////////////////////////////////////
void Blit_String(int x,int y,int color, char *string)
{ // функция отображает на экране передаваемую строку символов
// Расстояние между символами строки постоянно
// Для отображения символов вызывается функция blit_char
int index;
for (index=0; string[index]!=0; index++)
{
Blit_Char(x+(index<<3) ,y, string[index],color);
} // конец цикла for
} // конец функции
////////////////////////////////////////////////////////
void Delay(int t)
{
float x = 1;
while(t—>0)
x=cos(x);
} // конец функции
////////////////////////////////////////////////////////
void Set_Palette_Register(int index, RGB_color_ptr color) {
// функция устанавливает элемент таблицы цветов задаваемый
// параметром index. Значения компонент цвета задаются полями
// структуры color.
// указываем VGA-карте, что мы будем изменять регистр палитры
_outp(PALETTE_MASK,Oxff);
// указываем номер изменяемого регистра
_outp(PALETTE_REGISTER_WR, index);
// теперь меняем значения компонентов. Каждый раз используется
// один и тот же порт
_outp(PALETTE_DATA,color->red);
_outp(PALETTE_DATA,color->green);
_outp(PALETTE_DATA,color->blue);
} // конец функции
////////////////////////////////////////////////////////
void PCX_Init(pcx_picture_ptr image)
{
// функция выделяет память для загрузки PCX-файла
if ((image->buffer = (char far *)malloc (SCREEN_WIDTH * SCREEN_HEIGHT +1)));
printf("\ncouldn't allocate screen buffer");
} // конец функции
////////////////////////////////////////////////////////
void Plot_Pixel_Fast(int x,int y,unsigned char color)
{
// функция отображает на экране точку заданного цвета
```

```
// вместо умножения используется сдвиг
// пользуемся тем, что 320*y=256*y+64*y=y<<8+y<<6
video_buffer[ ((y<<8) + (y<<6)) + x] = color;
} // конец функции
//////////////////////////////////////////////////////////////////////////
void PCX_Delete(pcx_picture_ptr image)
{ // функция освобождает память, выделенную для загрузки PCX-файла
_ffree (image->buffer);
} // конец функции
//////////////////////////////////////////////////////////
void PCX_Load(char *filename, pcx_picture_ptr image, int enable_palette}
{ // функция загружает PCX-файл и заполняет поля структуры
// pcx_picture, включая изображение (после декомпрессии),
// заголовок и палитру
FILE *fp;
int num_bytes,index;
long count;
unsigned char data;
char far *temp_buffer;
// открыть файл
fp = fopen(filename,"rb");
// загрузить заголовок
temp_buffer = (char far *)image;
for (index=0; index<128; index++)
{
temp_buffer[index] = getc(fp);
} // конец цикла for
// загрузить данные и декодировать их в буфере
count=0;
while(count<=SCREEN_WIDTH * SCREEN_HEIGHT)
{ // получить первую часть данных
data = getc(fp);
// это RLE?
if (data>=192 && data<=255)
{ // сколько байт сжато?
num_bytes = data-192;
// получить значение цвета для сжатых данных
data = getc(fp);
// заполняем буфер полученным цветом
while(num_bytes-->0)
{
image->buffer[count++] = data;
} // конец цикла while
} // конец обработки сжатых данных
else
{
// поместить значение цвета в очередную позицию
image->buffer[count++] = data;
} // конец обработки несжатых данных
} // конец цикла while
// перейти в позицию, не доходя 768 байт от конца файла
fseek(fp,-768L,SEEK_END) ;
// загрузить палигру
for (index=0; index<256; index++)
{
// красная составляющая
image->palette[index].red = (getc(fp) >> 2);
// зеленая составляющая
image->palette[index].green = (getc(fp) >> 2);
// синяя составляющая
image->palette[index].blue = (getc(fp) >> 2) ;
} // конец цикла for
fclose(fp);
// если установлен флаг enable_palette, установить новую палитру
if (enable_palette)
{
for (index=0; index<256; index++)
```

```
{
Set_Palette_Register(index,
(RGB_color_ptr)&image->palette[index]);
} // конец цикла for

} // конец установки палитры
} // конец функции

////////////////////////////////////////
void PCX_Show_Buffer (pcx_picture_ptr image)
{ // функция копирует буфер, содержащий изображение из PCX-файла,
// в видеопамять
_fmemcpy(char far *)video_buffer,
 (char far *) image->buffer,SCREEN_WIDTH*SCREEN__HEIGHT);
} // конец функции
////////////////////////////////////////////////////
void Sprite_Init(sprite_ptr sprite, int x,int y, int ac, int as,int mc,int ms)
{
// функция инициализирует спрайт
int index;
sprite->x            = x;
sprite->y            = y;
sprite->x_old        = x;
sprite->y_old        = y;
sprite->width        = SPRITE_WIDTH;
sprite->height       = SPRITE_HEIGHT;
sprite->anim_clock   = ac;
sprite->anim_speed   = as;
sprite->motion_clock = me;
sprite->motion_speed = ms;
sprite->curr frame   = 0;
sprite->state        = SPRITE_DEAD;
sprite->num_frames   = 0;
sprite->background  = (char far *)malloc (SPRITE_WIDTH * SPRITE_HEIGHT+1);
// устанавливаем все указатели в значение NULL
for (index=0; index<MAX_SPRITE_FRAMES; index++) sprite->frames[index] = NULL;
} // конец функции
////////////////////////////////////////////////////////
void Sprite_Delete(sprite_ptr sprite)
{ // функция освобождает всю связанную со спрайтом память
int index;
_ffree(sprite->background) ;
// освобождаем память, выделенную под кадры анимации
for (index=0; index<MAX_SPRITE_FRAMES; index++) _ffree(sprite->frames(index]);
} // конец функции
////////////////////////////////////////////////////
void PCX_Grap_Bitmap (pcx_picture_ptr image, sprite_ptr sprite, int sprite_frame, int
grab_x, int grab_y)
{
// функция выделяет один кадр из буфера PCX-файла.
// Предполагается, что изображение размером 320x200 пикселей
// в действительности представляет собой массив 12x8 изображений
// каждое размерностью по 24x24 пикселя
int x_off,y_off, x,y, index;
char far *sprite_data;
//вначале выделяем память для размещения спрайта
sprite->frames[sprite_frame] = (char far *)malloc (SPRITE_WIDTH * SPRITE_HEIGHT);
// создаем альтернативный указатель для ускорения доступа
sprite_data = sprite->frames[sprite_frame];
// загружаем битовый образ в выделенную область памяти
// вначале вычисляем, какое из изображений копировать.
// Помните: в действительности PCX-файл представляет собой массив
// из отдельных элементов размером 24x24 пикселя.
// Индекс (0,0) соответствует левому верхнему изображению,
// (11,7) - правому нижнему.
x_off = 25 * grab_x + 1;
y_off = 25 * grab_y + 1;
```

```
// вычисляем начальный адрес
y_off = y_off * 320;
for (y=0; y<SPRITE__HEIGHT; y++)
{
for (x=0; x<SPRITE_WrETH; x++)
{
// берем очередной байт и помещаем его в текущую позицию буфера
sprite_data[y*24 + x] = image->buffer [y_off + x_off + x];
} // конец копирования строки
// переходим к следующей строке y_off+=320;
} // конец копирования
// увеличиваем счетчик кадров sprite->num_frames++;
} // конец функции
////////////////////////////////////
void Behind_Sprite(sprite_ptr sprite)
{
// функция сохраняет содержимое видеопамяти в той области,
// куда будет выведен спрайта
char far *work back;
int work_offset=0,offset,y;
// создаем альтернативный указатель для ускорения доступа
work_back = sprite->background;
// вычисляем смещение в видеопамяти
offset = (sprite->y << 6) + (sprite->y << 6) + sprite->x;
for (y=0; y<SPRITE_HEIGHT; y++)
{
// копируем очередную строку видеопамяти в буфер
_fmemcpy((char far *)&work_back[work_offset], (char far *)&video_buffer[offset],
SPRITE_WIDTH);
// переходим к следующей строке
offset      += SCREEN_WIDTH;
work_offset += SPRITE_WIDTH;
} // конец цикла for
} // конец функции
////////////////////////////////////////
void Erase_Sprite(sprite_ptr sprite)
{ // функция восстанавливает фон, сохраненный перед выводом спрайта
char far *work_back;
int work_offset=0,offset,y;
// создаем альтернативный указатель для ускорения доступа
work_back = sprite->background;
// вычисляем смещение в видеобуфере
offset = (sprite->y_old << 8) + (sprite->y_old << 6} + sprite->x_old;
for (y=0; y<SPRITE_HEIGHT; y++)
{
// копируем в видеопамять очередную строку буфера
_fmemcpy((char far *)&video_buffer [offset],(char far *)&work_back[work_offset],
SPRITE_WIDTH);
// перейти к следующей строке
offset      += SCREEN_WIDTH;
work_offset += SPRITE_WIDTH;
} // конец цикла for
} // конец функции
////////////////////////////////////////////////////
void Draw_Sprite(sprite_ptr sprite)
{
// функция, рисующая спрайт на экране вместо умножения
// используется сдвиг
char far *work sprite;
int work_offset=0,offset,x,y;
unsigned char data;
work sprite = sprite->frames[sprite->curr_frame];
// вычислить смещение спрайта в видеобуфере
offset = (sprite->y << 8) + (sprite->y << 6) + sprite->x;
for (y=0; y<SPRITE_HEIGHT;y++)
{
for (x=0; X<SPRITE_WIDTH; X++)
```

```
{
// проверяем, не является ли пиксель "прозрачным" (с кодом 0),
// если нет - рисуем
if ((data=work_sprite[work_offset+x]))
video_buffer[offset+x] = data;
} // конец вывода строки
// перейти к следующей строке в видеобуфере и буфере спрайта
offset      += SCREEN_WIDTH;
work_offset += SPRITE_WIDTH;
} //конец цикла no Y
} // конец функции
// ОСНОВНАЯ ПРОГРАММА ////////////////////////////////////////
void main(void)
{
long index,redraw;
RGB_color color;
int frame_dir = 1;
pcx_picture town, cowboys;
sprite cowboy;
// установить видеорежим 320x200x256
Set_Mode(VGA256);
// установить глобальные указатели на область памяти экрана
Set_Screen_Pointers();
// загрузить фон
PCX_Init((pcx_picture_ptr)&town) ;
PCX_Load( "town. pox", (pcx_picture_ptr)&town, 1) ;
PCX_Show_Buffer((pcx_picture_ptr)&town);
PCX_Delete((pcx_picture_ptr)&town);
// вывести на экран заставку игры
Blit_String(128, 24,50, "TOMBSTONE");
// загрузить образы
PCX_Init((pcx_picture_ptr)&cowboys) ;
PCX_Load("cowboys.pcx", (pcx_picture_ptr) &cowboys,0) ;
// извлечь все образы из PCX-файла
Sprite_Init((sprite_ptr)&cowboy,SPRITE_WIDTH,100,0,7,0,3) ;
PCX_Grap_Bitmap ((pcx_picture_ptr) &cowboys, (sprite_ptr) &cowboy, 0, 0, 0);
PCX Grap_Bitmap( (pcx_picture_ptr) &cowboys,
(sprite_ptr)&cowboy,1,1,0) ;
PCX_Grap_Bitmap((pcx_picture_ptr)&cowboys,
(sprite_ptr)&cowboy,2,2, 0) ;
PCX_Grap_Bitmap ((pcx_picture_ptr)&cowboys,
(sprite_ptr)&cowboy,3,1,0) ;
// очистить память, выделенную для загрузки PCX-файла
PCX_Delete((pcx_picture_ptr)&cowboys);
Behind_Sprite((sprite_ptr)&cowboy);
Draw_Sprite ((sprite_ptr) &cowboy);
// главный цикл
cowboy.state = SPRITE_ALIVE;
while (!kbhit()) {
redraw = 0; // используется как флаг необходимости перерисовки
if (cowboy.state==SPRITE_ALIVE)
{
//не пора ли менять кадр?
if (++cowboy.anim_clock > cowboy.anim_speed)
{
// сбрасываем счетчик времени
cowboy.anim_clock = 0;
if (++cowboy.curr_frame >= cowboy. num_frames)
{
cowboy.curr_frame = 0;
} // конец обработки достижения последнего кадра
redraw=1;
} // конец смены кадра
// проверяем не пора ли передвигать спрайт
if (++cowboy.motion_clock > cowboy.motion_speed)
{
// переустановить счетчик движения
```

```
cowboy.motion clock =0;
// сохранить старую позицию
cowboy.x_old == cowboy.x;
redraw =1;
//передвинуть спрайт
if (++cowboy.x >= SCREEN_WIDTH-2*SPRITE_WIDTH)
{
Erase_Sprite((sprite_ptr)&cowboy);
cowboy.state = SPRITE_DEAD;
redraw         = 0;
} // конец обработки достижения последнего кадра
} // конец обработки движения спрайта
} // конец обработки ситуации "ковбой жив"
else
{ // пытаемся "оживить" ковбоя
if (rand()%100 == 0 )
{
cowboy.state      = SPRITE_ALIVE;
cowboy.x          = SPRITE_WIDTH;
cowboy.curr frame = 0;
cowboy.anim.speed   = 3 + rand()%6;
cowboy.motion_speed = 1 + rand()%3;
cowboy.anim_clock   = 0;
cowboy.motion_clock = 0;
Behind_Sprite{(sprite_ptr)&cowboy);
}
} // конец процесса "оживления"
// теперь состояние спрайта изменено
if (redraw)
{
// удалить спрайт в старой позиции
Erase_Sprite((sprite_ptr)&cowboy) ;
// сохранить фон в новой позиции
Behind_Sprite((sprite_ptr)&cowboy) ;
// нарисовать спрайт в новой позиции
Draw_Sprite((sprite_ptr)&cowboy);
// обновить старые координаты
cowboy.x_old = cowboy.x;
cowboy.y_old = cowboy.y;
} // конец перерисовки
Delay(1000);
} // конец главного игрового цикла
for(index=0; index <300000;index++,Plot_Pixel_Fast(rand()%320,
                                        rand()%200,0));
//Перейти обратно в текстовый режим
Set_Mode(TEXT_MODE);
}//Конец функции main
```

Итог

Эта глава, наверное, самая важная в книге. Среди всей информации, которая, может быть, не относится напрямую к играм, вы узнали общую концепцию построения видеоигр, что абсолютно необходимо для их написания. Так что, если вы чего-то не поняли, то остановитесь и прочтите главу вновь (не беспокойтесь, я вас подожду).

Если вы прочитали и поняли в этой главе все, то .должны знать, что:

- Мы изучили VGA-карту и ее архитектуру;
- Мы узнали о режиме l3h, который дает лучшее разрешение и наиболее прост для программирования;
- Мы узнали, как в режиме l3h программировать регистры цвета, рисовать пиксели, загружать PCX-файлы и перемещать битовые образы;
- Мы поговорили о спрайтах и создали простую библиотеку для работы с ними;
- Мы затронули довольно сложные вещи, такие как дублирующий буфер и
- вертикальная синхронизация, о которых более подробно мы поговорим чуть позже;
- И, наконец, собрав все вместе, мы сделали демонстрационную программу, рисующую маленького ковбоя, который ходит по городу.

До встречи в новой главе.

ТРЕТЬЕ ИЗМЕРЕНИЕ

Последние достижения в области электроники и программирования приблизили наши фантазии о трехмерных видеоиграх к осязаемой реальности. Разработчики игр сегодня могут создать трехмерные игры для недорогого ПК, позволяющие игрокам получить ощущения, всего несколько лет назад казавшиеся недоступными,

Говоря по правде, и в прошлом были трехмерные имитаторы полетов и несколько пространственных игр. Но все они были очень медленными и нуждались в мощных компьютерах. Естественно, это резко сужало круг потенциальных игроков. Все эти ограничения заставляли программистов искать пути создания нетрадиционных методов рен-деринга. Эти новые способы были очень просты и имели производительность, еще пару лет назад считавшуюся невозможной.

В данной главе мы сначала обсудим традиционные методы рендеринга для ПК. а затем погрузимся в технику, называемую «отсечение лучей». Этот метод осуществляет рендеринг со скоростью, близкой к световой...

Из этой главы вы узнаете:

- Что такое трехмерное пространство
- Как изображаются точки, многоугольники и объекты в трехмерном пространстве;
- Масштабирование, трансляция и повороты в трехмерном пространстве;
- Проекции;
- Геометрическое моделирование;
- Алгоритм удаления невидимой поверхности;
- Алгоритм художника;
- Алгоритм, использующий Z-буфер;
- Что такое текстуры;
- Метод трассировки лучей;
- Алгоритм отсечения лучей;
- Реализация алгоритма отсечения лучей;
- Усовершенствование алгоритма отсечения лучей;
- Освещение, затемнение и цветовая палитра.
-

Мы начнем наш урок с основных концепций трехмерного пространства, чтобы кое-что вспомнить и понять. Затем мы бросимся вперед, чтобы узнать как создаются игры типа Wolfenstein, DOOM и Terminator Rampage!

Что такое трехмерное пространство

Трехмерное пространство... Звучит как строка из фантастического рассказа... Ну очень похоже. Трехмерное пространство - это просто расширение двухмерной плоскости. Надо сразу отметить, что рендеринг трехмерной графики весьма сложен. Вообще же, сложность рендеринга экспоненциально возрастает с добавлением новых измерений и происходит это оттого, что и сами образы при этом усложняются. Наша задача заключается в том, чтобы понять, как работать с новым измерением. Мы должны все это изучить.

Говоря языком математики, любая точка в трехмерном пространстве описывается с помощью уникального набора трех координат: x, y и z. Как мы уже обсуждали ранее, обычно экран представляется плоскостью X и Y, а координата z перпендикулярна экрану.

В отличие от плоскости, где x-координата горизонтальна, а y-координата вертикальна, трехмерные системы координат бывают двух типов:

- Левосторонняя система координат;
- Правосторонняя система координат.

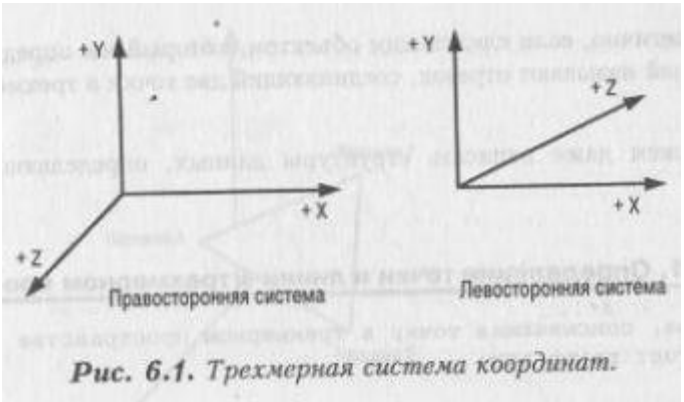


Рис. 6.1. Трехмерная система координат.

На рисунке 6.1 показано представление обоих способов отображения трехмерных систем.

В дальнейшем для всех наших рассуждений, примеров и программ мы будем использовать только правостороннюю' систему координат. На то есть две причины:

- Правосторонняя система удобней в работе, поскольку в ней проще выполняется визуализация;

- Правосторонняя система распространена как стандарт.

Конечно, вы можете сказать, что экран компьютера — это плоскость, и мы не можем преобразовывать трехмерные образы в двух измерениях. Верно. Вы абсолютно правы. Но у нас есть возможность отобразить их на плоскость. Мы даже можем видеть «тени» объектов. И сделать это позволит проекция. При этом модель выглядит на двухмерном экране так, что у зрителей возникает полное ощущение объемности. Такие игры как DOOM и Wolfenstein трехмерны только в нашем восприятии. Можно сказать, что в них смоделирован особый случай трехмерного пространства, образы которого можно обрабатывать гораздо проще и быстрее.

В любом случае, мы еще к этому вернемся, а теперь давайте поговорим об основных понятиях трехмерного пространства.

Точки, линии, многоугольники и объекты в трехмерном пространстве

Как мы уже видели, точка в трехмерном пространстве имеет три координаты (x,y,z). Этой информации достаточно, чтобы ее однозначно определить в пространстве.Будет логично, если следующим объектом, который мы определим, станет линия. Линией называют отрезок, соединяющий две точки в трехмерном пространстве. Мы можем даже написать структуры данных, определяющие точку и линию.

Листинг 6.1. Определение точки и линии в трехмерном пространстве.

```
// структура, описывающая точку в трехмерном пространстве
typedef struct point_typ
{
float x,y,z                 // координаты точки
} point, *point_ptr;
// структура, описывающая линию в трехмерном пространстве
typedef struct line_typ
{
point start, end;        // линия задается двумя точками
} line, *line_ptr;
```

Используя структуры из Листинга 6,1, давайте определим линию, которая начинается в точке (0,0,0) и идет в точку (100,200,300)

```
line linel;
linel.start.x = 0;
linel.start.y= 0;
linel.start.z = 0;
linel.end.x = 100;
linel.end.y = 200;
linel.end.z = 300;
```

Теперь мы имеем описание линии. Если мы захотим, то сможем создать трехмерный мир, состоящий из линий и точек, но это будет скучным и тоскливым занятием.

Нам нужен больший уровень абстракции для моделирования объектов, и для этого нам пригодятся многоугольники. Как вы поняли из четвертой главы, многоугольник - это множество вершин, соединенных отрезками прямых. Вершины определяют границы многоугольника. В трехмерном пространстве Многоугольники очень похожи на своих двухмерных собратьев. Попробуем определить трехмерный треугольник. Он может выглядеть так, как это изображено иа рисунке 6.2.

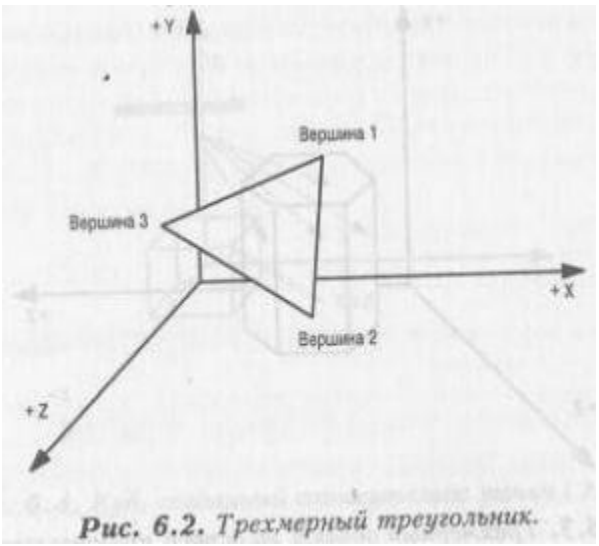


Рис. 6.2. Трехмерный треугольник.

Как вы можете видеть, на листе бумаги весьма несложно представить трехмерный объект. Мы будем использовать для описания «диагональный вид». Позже,мы к этому еще вернемся, а

сейчас важно понять идею.

Описать многоугольник довольно просто: мы применим старое определение многоугольника и просто добавим к нему несколько атрибутов для создания новой законченной структуры. В Листинге 6.2 показана такая структура.

Листинг 6.2. Определение трехмерного многоугольника.

```
// структура, описывающая многоугольник
typedef struct polygon_typ
{
int num_vertices;                // число вершин
vertices[MAX VERTICES];          // координаты вершин
int color;                       // цвет многоугольника
}polygon, *polygon_ptr;
```

Как можно заметить, в структуре описаны вершины и цвета. Эти составляющие необходимы для правильного отображения. Теперь, когда у нас есть структура, описывающая многоугольник, следующим шагом будет определение объекта на основе многоугольников. На рисунке 6.3 продемонстрирован один из таких объектов.

Теперь мы можем добавить еще один уровень к нашему описанию. Объект - это набор многоугольников. Создадим структуру, которая бы поддерживала эту концепцию:

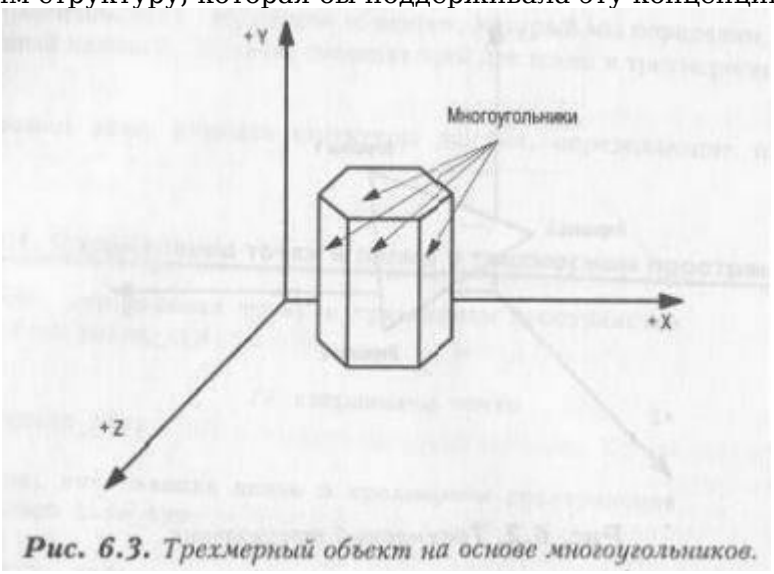


Рис. 6.3. *Трехмерный объект на основе многоугольников.*

Листинг 6.3. Описание трехмерного объекта на основе многоугольников.

```
// структура, описывающая объект
typedef struct object_typ
{
int num_faces;              // число граней
polygon faces[MAX_FACES]; // грани, представленные многоугольниками
float xo,yo,zo;             // координаты объекта в пространстве
int visible;                // виден ли объект на экране?
} object, *object_ptr;
```

Структура данных в Листинге 6.3 описывает объект, который образован , множеством многоугольников или поверхностей. Используя эти структуры данных и определения, мы можем создать несколько трехмерных объектов: космический корабль, планету и окружающее космическое пространство.

Чтобы поместить объекты в трехмерное пространство, мы должны знать их пространственное расположение. То есть мы должны определить значения xo, yo и zo для каждого предмета. Так же, как и в случае с двухмерными объектами (которые мы уже обсуждали в четвертой главе), пространственные объекты мы будем определять в собственных локальных системах координат (0,0,0). Затем, когда мы будем перемещать объект, мы его просто переведем в конечную позицию.

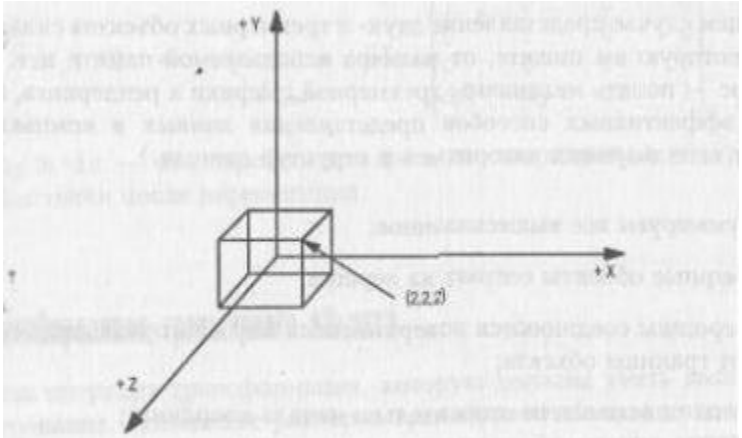


Рис. 6.4. Куб, созданный относительно точки (2,2,2).

Для наших структур это будет точка (xo,yo,zo). Решением этой задачи будет простой перенос каждой из точек объекта, так же, как мы это делали для двухмерных объектов. Мы можем проверить этот метод и для объемных фигур. Например, представим себе куб, с вершиной в точке (2,2,2) (см. рис. 6.4). Если мы посмотрим на куб, то увидим, что он состоит из восьми вершин и шести поверхностей. Используя наши структуры данных, мы можем описать куб как объект с шестью гранями. Проблема, возникающая в данном случае, состоит в том, что это не самый лучший способ описания объекта. Ведь любая поверхность ограничена четырьмя точками и каждая из этих точек является общей еще для двух поверхностей. Это значит, что описание избыточно.

Возможно, более удачной окажется структура данных, содержащая список вершин. В этом случае избыточности не возникает. Однако при этом структура станет более общей и сложной, поскольку:

- Мы должны будем иметь указатели либо индексы, или то и другое вместе для ссылки на вершины, необходимые для построения геометрической фигуры. Это увеличивает время распознавания данных объектов;
- Наши структуры могут использовать заранее определенные массивы для хранения вершин и многоугольников. Это неэффективно использует память. Массивы должны быть одного размера, так как, независимо от того, используем ли мы один элемент массива или весь массив, нам необходимо отводить место под максимальное число элементов.

Эти факты надо принимать во внимание, когда создаете структуры для трехмерных объектов. Таким образом, для наших целей структуры данных из Листингов 6.2 и 6.3 являются наиболее простыми для работы. Если же вы хотите создать набор реальных трехмерных структур, то должны использовать другую тактику.

В общем случае представление двух- и трехмерных объектов сильно зависит от игры, которую вы пишете, от размера используемой памяти и т. д. (Наша цель сейчас - понять механизмы трехмерной графики и рендеринга, а не поиск наиболее эффективных способов представления данных в компьютере. Это зависит от используемых алгоритмов и структур данных.)

Просуммируем все вышесказанное:

- Трехмерные объекты состоят из вершин;
- Эти вершины соединяются поверхностями или многоугольниками, которые задают границы объекта;
- Объекты описываются относительно начала координат;
- Существует много способов представления трехмерных объектов и вы должны выбрать тот, который устраивает вас по скорости и объему памяти.

Перемещения, масштабирование и повороты в трехмерном пространстве

Все объекты в видеоиграх представлены точками, линиями и геометрическими фигурами. Потому мы должны всегда производить их разбиение на исходные составляющие — точки. Только проделав это, мы можем их преобразовывать. Поскольку точки — это вершины объектов, то они могут рассматриваться вообще отдельно от объекта, но при этом представлять его в целом. Например, если мы хотим повернуть или переместить куб, то, прежде чем выполнять данное преобразование, нам нужно разбить объект на многоугольники, а затем на точки.

Понимая это, мы должны сосредоточить свое внимание на способах преобразования точек.

Перемещение трехмерного объекта

Для перемещения точки (x.y.z) на расстояние (dx,dy,dz) необходимо выполнить следующие операции:

```
x=x+dx;
y=y+dy;
z=z+dz;
```

Если мы хотим использовать эту матрицу, то должны представить точку в виде четырех

компонентов (x,y,z, 1). Матричное умножение будет выглядеть так:

$$[x' \quad y' \quad z' \quad 1] = [x \quad y \quad z \quad 1] \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{vmatrix}$$

где dx, dy и dz - это перемещения по осям координат, а x', y' и z' - координаты точки после перемещения.

Масштабирование трехмерного объекта

Следующая операция трансформации, которую должны уметь выполнять, это масштабирование. Изменение размеров трехмерного объекта похоже на двухмерное масштабирование. Здесь показано масштабирование точки (x,y,z) с коэффициентом S:

```
x=x * S;
y=y * S;
z=z * S;
```

Все очень просто. Только кажется, что трехмерная графика сложна для понимания. Для описания преобразований с помощью матриц, мы опять должны представить точку в виде (x,y,z,1):

$$[x' \quad y' \quad z' \quad 1] = [x \quad y \quad z \quad 1] \begin{vmatrix} S & 0 & 0 & 0 \\ 0 & S & 0 & 0 \\ 0 & 0 & S & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Если вы решите масштабировать каждый из компонентов по-разному, то вам потребуются разные коэффициенты для каждого измерения:

$$[x' \quad y' \quad z' \quad 1] = [x \quad y \quad z \quad 1] \begin{vmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Это приведет к неоднородному масштабированию.

Вращение трехмерного объекта

Последняя трансформация, о которой мы поговорим - это вращение. Вращение трехмерного объекта аналогично двухмерной ротации; необходимо только добавить третье измерение.

Вращение, параллельное оси X

Следующая матрица преобразований вращает точку (x,y,z) параллельно оси X:

$$[x' \quad y' \quad z' \quad 1] = [x \quad y \quad z \quad 1] \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos r & \sin r & 0 \\ 0 & -\sin r & \cos r & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

где r — угол вращения в радианах.

Вращение, параллельное оси Y

Матрица преобразования, вращающая точку параллельно оси Y:

$$[x' \quad y' \quad z' \quad 1] = [x \quad y \quad z \quad 1] \begin{vmatrix} 0 & 1 & 0 & 0 \\ \sin r & 0 & \cos r & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

где r — угол вращения в радианах.

Вращение, параллельное оси Z

Матрица преобразования, вращающая точку параллельно оси Z;

$$[x' \quad y' \quad z' \quad 1] = [x \quad y \quad z \quad 1] \begin{vmatrix} -\sin r & \cos r & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

где r — угол вращения в радианах.

Последнее слово о трехмерных трансформациях

Подведем итог разговору о трансформациях трехмерных объектов. Реализовать эти трансформации не сложно. Это только часть возможных преобразований. Существует много вариаций и несколько совершенно новых типов — деформация и т. д. В общем, мы уже достаточно знаем, чтобы перейти к проекциям.

Проекции

Сейчас мы знаем, как изобразить трехмерный объект и как произвести операции перемещения, масштабирования и поворота этого объекта. Возникает вопрос: «А как мы можем рисовать трехмерные объекты на плоском экране?» ответ прост: мы «проецируем» их на поверхность экрана.

Проекция трехмерных объектов на плоскость довольно проста и дает хорошие результаты. К сожалению, образы при этом не всегда выглядят реалистично. Ваши глаза имеют одну точку зрения, а, следовательно, образ будет похож на трехмерный, но не станет «по-настоящему» объемным.

Замечание

Существуют специальные шлемы с дисплеями, дающие стереоскопический эффект. Эта шлемы используются в системах виртуальной реальности. Проблема, возникающая с их использованием, заключается в том, что параллакс или фокальная точка различна для каждого человека. Если шлем не отрегулирован и точка не подобрана, то зритель вскоре получит сильнейшую головную боль.

Тем не менее, мы будем использовать монитор для проецирования трехмерного образа на экран. Я хотел бы обсудить типы проекции, которые могут применяться с этой целью: параллельная, или ортогональная проекция и перспективная проекция. Рисунок 6.5 показывает диаграммы каждого типа проекций.

Параллельная проекция проста в реализации, но образы не выглядят объемными. Они, скорее, похожи на обычные плоские картинки. Для реализа ции такой проекции достаточно убрать Z-компонент каждой точки трехмерного объекта и затем нарисовать объект, как двухмерный.

С другой стороны, перспективная проекция дает большее приближение и выглядит почти «трехмерно». Она имеет качество «длинной дороги». На рисунке 6.6 изображена такая «дорога». Перспективная проекция принимает во внимание 2-компонент и соответственно изменяет компоненты X и Y.

Элементы, которые подвергаются «перспективному» преобразованию, должны:

- Просто делиться или умножаться на Z-компонент;
- Обладать дистанцией просмотра.

Вскоре мы затронем детали проецирования, но сначала давайте поговорим об экране и его взаимоотношениях с координатами трехмерного пространства.

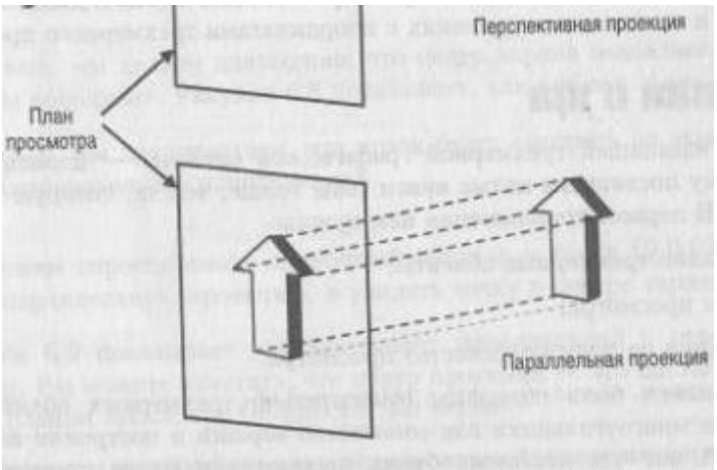


Рис. 6.5. Два типа трехмерной проекции.

Рис. 6.6. *Дорога в никуда.*

Копирование на экран

Создание настоящей трехмерной графической системы — довольно сложное дело. Этому посвящены целые книги (еще толще, чем та, которую вы держите в руках). В первом приближении нам нужны:

- Кое-какие трехмерные объекты;
- Объем просмотра;
- Проекция на план (плоскость) просмотра.

Мы можем быть спокойны относительно трехмерных объектов — мы определили многоугольники как множество вершин и построили на их основе объекты. А вот что касается объема просмотра и плана просмотра, то тут требуются некоторые разъяснения.

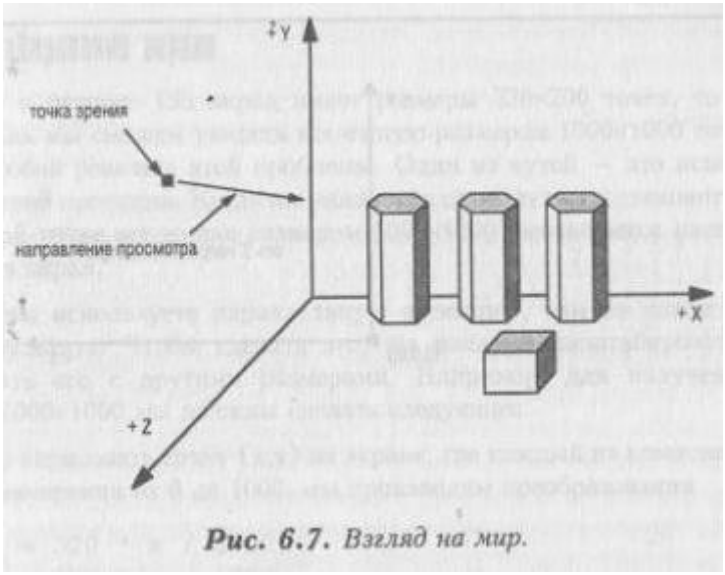


Рис. 6.7. *Взгляд на мир.*

Когда мы смотрим на трехмерный мир, то сами должны в нем где-то находиться и иметь направление просмотра. Рисунок 6.7 иллюстрирует эту мысль.

Учитывая, что в реальном мире мы смотрим на объекты с определенной точки и в определенном направлении, мы можем их построить на экране, который и будет являться планом просмотра. Такой подход довольно сложен, поскольку требует выполнения множества преобразований для решения данной задачи в общем виде. Но, поскольку мы программисты, нас не очень должно волновать общее решение задачи. Нам надо создавать игры, выглядящие как трехмерные.

- Во-первых, мы делаем допущение, что центр экрана совпадает с центром системы координат. Рисунок 6.8 показывает, как она выглядит на экране;
- Во-вторых, мы предполагаем, что игрок будет смотреть на экран вниз по оси Z (отрицательное направление).

Мы можем спроецировать одиночный пиксель в точку (0,0,0) на экран, используя параллельную проекцию, и увидеть точку в центре экрана. Рисунок 6.9 показывает разницу между параллельной и перспективной проекциями. Вы можете заметить, что центр проекции - это дистанция между игроком и планом просмотра (поверхностью экрана).

Математика для проецирования трехмерных объектов на экран сказочно проста и мы ее реализуем за пару минут. Но пока поговорим о масштабировании.

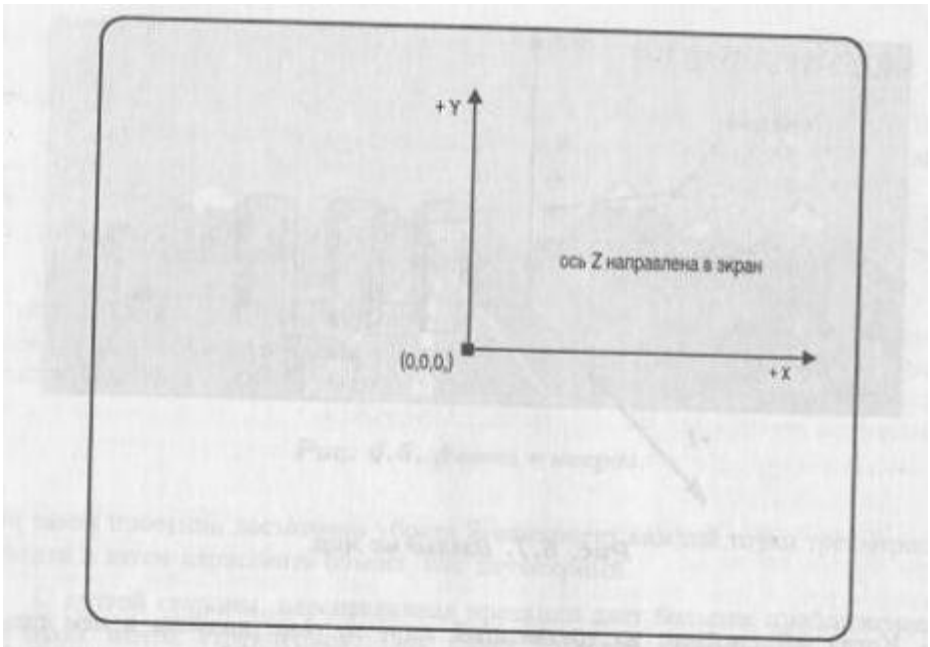


Рис. 6.8. *Отображение правосторонней системы координат на экран.*

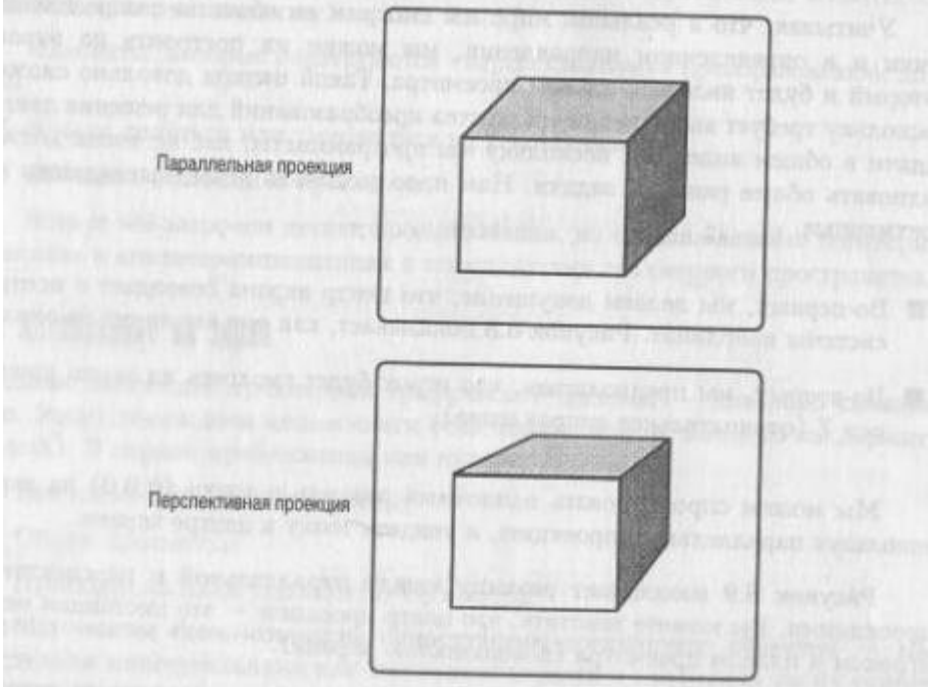


Рис. 6.9. *Два типа проекций на экран.*

Масштабирование экрана

Поскольку в режиме 13h экран имеет размеры 320x200 точек, то возникает вопрос: «Как мы сможем увидеть вселенную размером 1000x1000 точек?» Есть много способов решения этой проблемы. Один из путей - это использование перспективной проекции. Когда мы удаляемся от объекта, он становится меньше. В некоторой точке вселенная размером 1000x1000 уменьшается настолько, что «влезает» в экран.

Если вы используете параллельную проекцию, вам не удастся добиться такого результата. Чтобы сделать это, вы должны масштабировать экран и использовать его с другими размерами. Например, для получения экрана размером 1000x1000 мы должны сделать следующее.

Чтобы нарисовать точку (x,y) на экране, где каждый из компонентов имеет диапазон измерения от 0 до 1000, мы производим преобразования

```
x_screen = 320 * x / 1000;
y_screen = 200 * y /1000;
```

где x_screen и y_screen будут истинными координатами пикселя на экране размером 320x200.

Конечно, изменив таким образом масштаб экрана, вы не получите разрешения 1000x1000 точек. Разрешающая способность останется прежней - 320x200. Более того, некоторые точки будут изображены в одних и тех же местах. Например, точки с координатами 999 и 1000 сольются в одну,

и вы не сможете их различить. Но это и не столь важно, поскольку трехмерное изображение на основе многоугольников отличается от битовых изображений двухмерного мира.

Математические основы параллельных проекций

Математика, необходимая для произведения параллельной проекции, элементарна - вы просто убираете z-координату и рисуете каждую точку объекта по координатам (x,y). Вот и вся трансформация. Для примера возьмем точку (x,y,z).

```
x_parallel = x
y_parallel= y
plot x,y
```

Математические основы верспективной проекции

Произведение перспективной трансформации не многим отличается от параллельной проекции. Мы используем z-компонент для масштабирования координат x и y с целью «отдаления» точки от плана просмотра (поверхности экрана), и для придания объекту более реалистичного вида.

Рассмотрим эту операцию на примере: дана точка (x,y,z), удаленная от плана просмотра на расстояние D:

```
x_perspective = D * x / z
y_perspective = D * y / z
```

Вот и все. Достаточно умножить каждый компонент на расстояние и разделить на значение Z-координаты. Образ получается похожим на трехмерный.

Объем просмотра

Мы говорили об объектах и проекциях в трехмерном пространстве, но не упоминали об *объеме просмотра.* Это понятие подобно области отсечения для двухмерного мира (мы это обсуждали в четвертой главе, «Механизмы трехмерной графики»). Так же, как мы отсекаем объекты у границ экрана в двухмерных играх, мы должны проделывать это и с трехмерными объектами. Видимая часть пространства имеет шесть поверхностей и чем-то напоминает трехмерную трапецию или усеченную пирамиду (см. рис. 6.10). Эта область иногда называется еще *телесным углом просмотра.*

Рис. 6.10. Объем просмотра.

В общем случае, нам нужно отсечь каждый объект или его часть, выходящую за границы телесного угла просмотра. Поскольку отсекаемые фрагменты при перспективной проекции становятся не видны, то и нет необходимости выполнять их рендеринг.

Наиболее важны ближняя и дальняя плоскости отсечения (рис, 6.10). Когда объект подходит слишком близко к экрану, то он должен исчезать (он как-бы оказывается у вас за спиной), а когда он удаляется на слишком большое расстояние, то мы его тоже можем убрать, поскольку он просто превратится при рендеринге в точку.

Я больше ничего не буду говорить про трехмерное отсечение - оно не потребуется для игр типа DOOM и Wolfenstem, но может пригодиться для создания различных «леталок» и «ездилок». Вы можете использовать ту же тактику, что и для двухмерного отсечения. Теперь нам надо познакомиться еще с одной вещью — моделированием цельных геометрических объектов.

Геометрическое моделирование

Геометрическое моделирование необходимо для придания играм большего реализма. Объекты, построенные компьютером при геометрическом моделировании, выглядит цельным. Посмотрите на рисунок 6.11, чтобы увидеть разницу между цельной геометрической и каркасной

моделями.

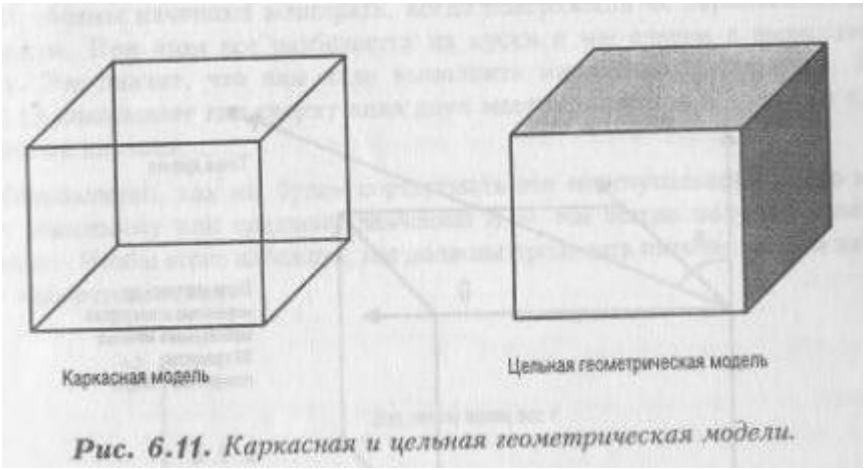


Рис. 6.11. Каркасная и цельная геометрическая модели.

Как видите, геометрическая модель выглядит более убедительно. Сегодня использование каркасных объектов уже нельзя назвать удовлетворительным решением. Теоретически создание цельных моделей ненамного сложнее, нежели каркасных. Мы можем использовать те же самые структуры данных, преобразования и проекции. Но когда мы пытаемся нарисовать цельный трехмерный объект, возникают определенные проблемы. Как вы понимаете, существуют еще и скрытые поверхности — те, которые не видны игроку (например, тыльная сторона рассматриваемого объекта) и которые не должны появляться на экране. Компьютер не понимает разницы между видимыми и невидимыми поверхностями. Он нарисует их, все. Чтобы избежать этого, мы должны найти способы удаления скрытых поверхностей. Сейчас мы об этом и поговорим.

Удаление невидимых поверхностей

Среди программистов, работающих в области компьютерной графики, техника удаления невидимых поверхностей считается «высшим пилотажем». Известно, что удаление невидимых поверхностей является сложной математической задачей. Существует довольно много алгоритмов для ее решения, но все они очень сложны в реализации и редко работают с приемлемой для видеоигр скоростью.

Удаление невидимых поверхностей можно разделить на две фазы:

Фаза 1. Прежде всего удаляются поверхности, которые никогда не будут видны с точки зрения наблюдателя. Для этого мы должны использовать точку пересечения между вектором наблюдения и вектором нормали к каждой из рассматриваемых плоскостей. Мы вычисляем это значение. Если значение угла меньше 90°, то поверхность видна, если больше 90° - нет и она удаляется.

Фаза 2. После того как скрытые поверхности удалены, видимые плоскости должны быть окрашены. Вы должны быть уверены, что в течение этой фазы, объекты будут выглядеть правильно.

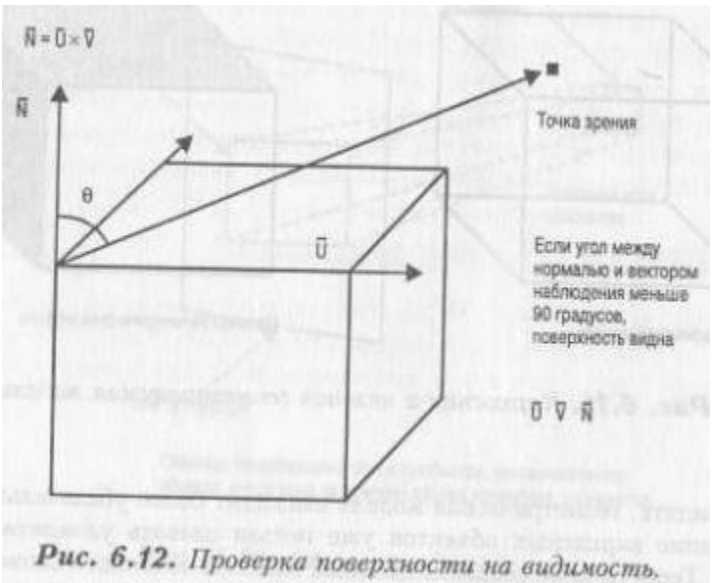


Рис. 6.12. Проверка поверхности на видимость.

Существует популярный Алгоритм Художника, который это хорошо умеет делать. Он работает, выполняя пять тестов для каждой видимой пары многоугольников (поверхностей) и затем создает последовательность их окраски от дальней части изображения к ближней.

Другая техника создания этой последовательности носит название Алгоритма Z-буфера. Он работает в пространстве образа на уровне пикселей. Он прост в реализации, но медленно работает и требует много памяти.

Настоящие разработчики видеоигр никогда не используют эти алгоритмы в чистом виде. Мы должны создать свою систему», которая будет сочетать оба метода. Для этого детально рассмотрим каждый из них.

Алгоритм художника

Алгоритм Художника — это один из тех алгоритмов, которые могут создавать ощущение реальности. Основная идея Алгоритма Художника состоит в сортировке поверхностей таким образом, что при рендеринге это выглядит корректно. Наиболее просто этот алгоритм может быть реализован, когда каждая поверхность параллельна плану просмотра (то есть перпендикулярна лучу зрения). В этом случае нам достаточно отсортировать поверхности в порядке уменьшения значения координаты Z. Затем мы сначала нарисуем самую дальнюю поверхность, потом более близкую и т, д. Это достаточно специфичное условие, но и у него есть свои реализации.

Проблемы начинают возникать, когда поверхность не параллельна плану просмотра. При этом все/разбивается на куски и мы вязнем в выполняемых тестах. Это значит, что нам надо выполнить множество вычислений. Рисунок 6.13 показывает вид сверху вниз двух многоугольников в одном из самых неудачных случаев.

Безразлично, как мы будем сортировать эти многоугольники - по минимуму, максимуму или среднему значению Z - мы всегда получим неверный результат. Чтобы этого избежать, мы должны проделать пять тестов для каждой пары многоугольников.

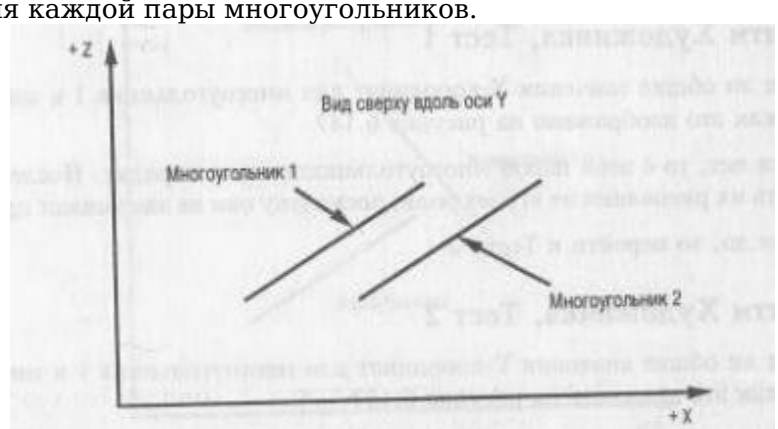


Рис. 6.13. Неудачное расположение двух многоугольников.

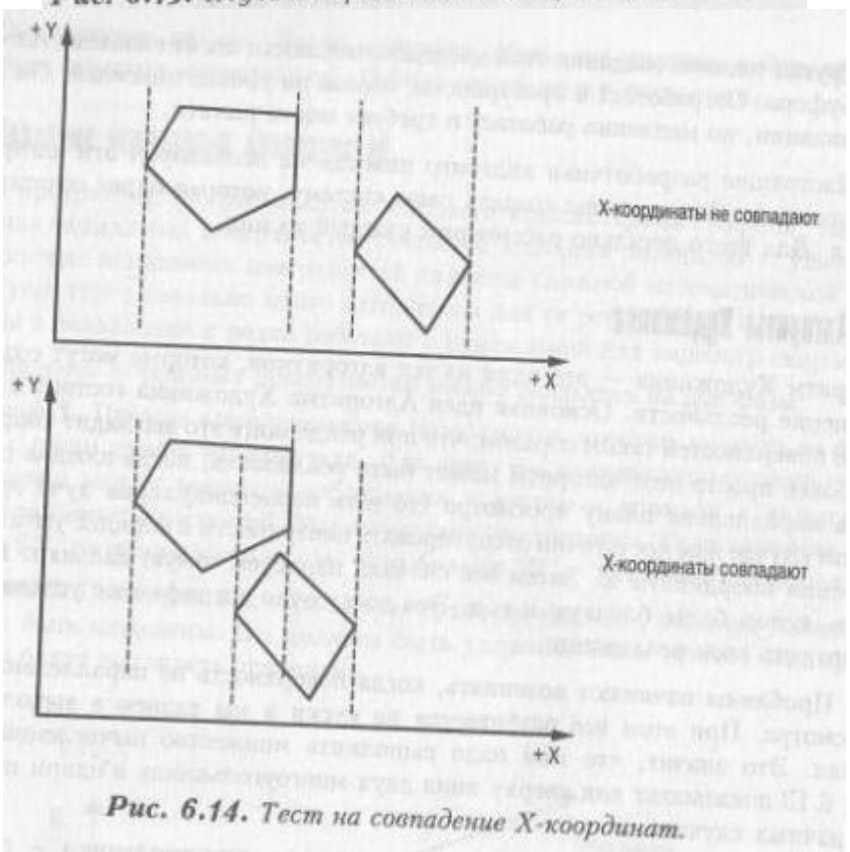


Рис. 6.14. Тест на совпадение X-координат.

Тесты выполняются только в том случае, если значения Z для двух многоугольников совпадают. Иначе, они могут рисоваться в любой последовательности.

Алгоритм Художника, Тест 1

Имеются ли общие значения X-координат для многоугольника 1 и многоугольника 2, как это изображено на рисунке 6.14

- Если нет, то с этой парой многоугольников все в порядке. Последовательность их рисования не играет роли, поскольку они не закрывают друг друга;
- Если да, то перейти к Тесту 2.

Алгоритм Художника, Тест 2

Имеются ли общие значения Y-координат для многоугольника 1 и многоугольника 2, как это показано на рисунке 6.15?

- Если нет, то их можно рисовать также в любом порядке;
- Если да, то выполнить Тесты 3 и 4.

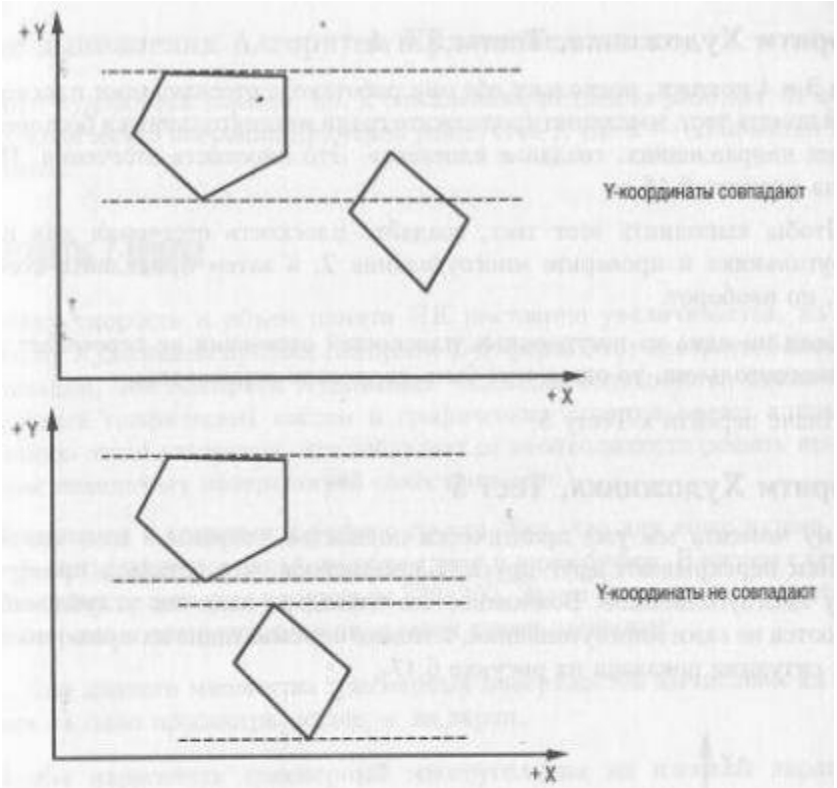


Рис. 6.15. *Тест на совпадение Y-координат.*

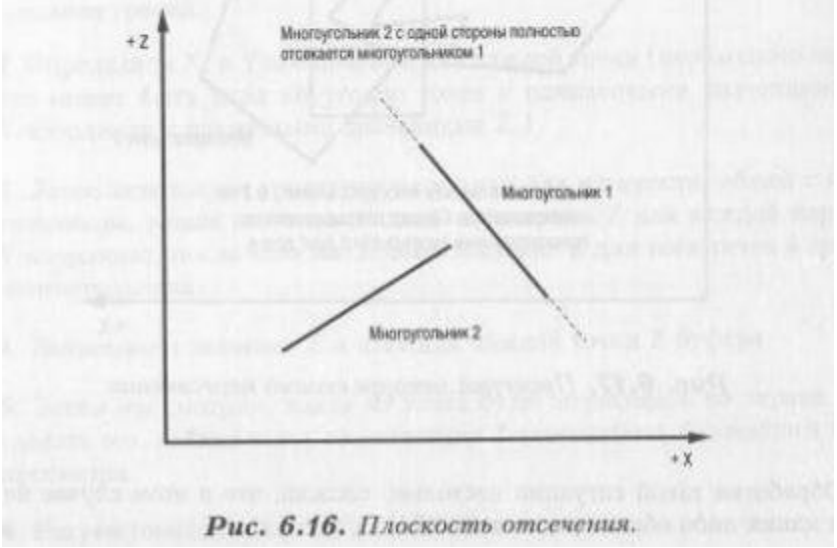


Рис. 6.16. *Плоскость отсечения.*

Алгоритм Художника, Тесты 3 и 4

Тесты 3 и 4 похожи, поскольку оба они работают с отсекающими плоскостями. Чтобы понять тест, мысленно продолжите грани многоугольника в бесконечность в обоих направлениях, создавая плоскость. Это плоскость отсечения. Посмотрите на рисунок 6.16.

Чтобы выполнить этот тест, создайте плоскость отсечения для первого многоугольника и проверьте многоугольник 2, а затем проделайте все то же самое, но наоборот.

- Если ни одна из построенных плоскостей отсечения не пересекает другой многоугольник, то они могут быть корректно отрисованы;
- Иначе перейти к Тесту 5.

Алгоритм Художника, Тест 5

К этому моменту мы уже практически полностью уверены в том, что многоугольники перекрывают друг друга. Единственное, что осталось проверить — форму многоугольников. Возможно, что благодаря наличию углублений перекрываются не сами многоугольники, а только описывающие их прямоугольники. Такая ситуация показана на рисунке 6.17.

Рис. 6.17. *Проверка некорректного пересечения.*

Обработка такой ситуации настолько сложна, что в этом случае не существует каких-либо общих рекомендаций,

Время выполнения Алгоритма Художника

Алгоритм художника хорошо, но, к сожалению, медленно работает. В худшем случае, количество операций примерно равно $O(n^2)$, где n - количество многоугольников.

Алгоритм Z - буфера

Поскольку скорость и объем памяти ПК постоянно увеличивается, на смену Алгоритму Художника пришел Алгоритм Z-буфера. Этот алгоритм более прост в реализации, чем Алгоритм Художника. (Сегодня большинство высокопроизводительных графических систем и графических станций имеют аппаратную реализацию этого алгоритма, что избавляет от необходимости решать проблему удаления невидимых поверхностей самостоятельно).

Реализация Алгоритма Z-буфера проста. Все, что для этого нужно - сам 2-буфер, который имеет такой же объем, как и видеобуфер. В нашем случае это будет матрица целых чисел размером 320x200. Затем мы заполняем ее значениями Z-координат многоугольников, следуя таким правилам:

1. Для данного множества трехмерных поверхностей вычисляем их проекции на план просмотра, иначе - на экран. Чтобы нарисовать трехмерный многоугольник на плоском экране, мы должны спроецировать его на план просмотра, используя один из двух видов проекции, которые мы обсуждали ранее. После проецирования многоугольник будет обладать множеством вершин, являющихся точками на плоскости. Потом мы заполним многоугольник, используя эти точки для создания граней.
2. Определяем X- и Y-компоненты для каждой точки (необходимо помнить, что может быть сколько угодно точек с одинаковыми значениями X- и Y-координат и различными значениями Z.)
3. Затем используем уравнение плоскости для плоскости, общей с многоугольником, решая его относительно компонента Z для каждой пары X- и Y-координат, после чего вычисляем

значение Z для всех точек в границах многоугольника.

4. Записываем значение Z и цвет для каждой точки Z-буфера.
5. Затем мы смотрим, какая из точек будет нарисована на экране. Чтобы сделать это, найдем точку со значением Z-координаты, ближайшей к плану просмотра.
6. Рисуем пиксель с цветом данной точки.

Уравнение плоскости

Я говорил, что мы можем использовать уравнение плоскости для многоугольника, для нахождения значения Z-компонента каждого из пикселей внутри преобразуемого прямоугольника. Вот это уравнение,

Дано: Точка (x,y) и вектор нормали к многоугольнику <Nx,Ny,Nz>

$$Z = \frac{Nz}{1 - Nx * X - Nv * Y}$$

Использование уравнения плоскости для вершин многоугольника

А каким образом мы можем составить уравнение плоскости, зная только вершины многоугольника? Очень просто: так как все вершины прямоугольника принадлежат одной плоскости, мы можем взять две смежные вершины и построить к ним вектор нормали. Рисунок 6.18 показывает, как это сделать.

Вектор нормали может быть использован в уравнении плоскости для вычисления Z-компонента.

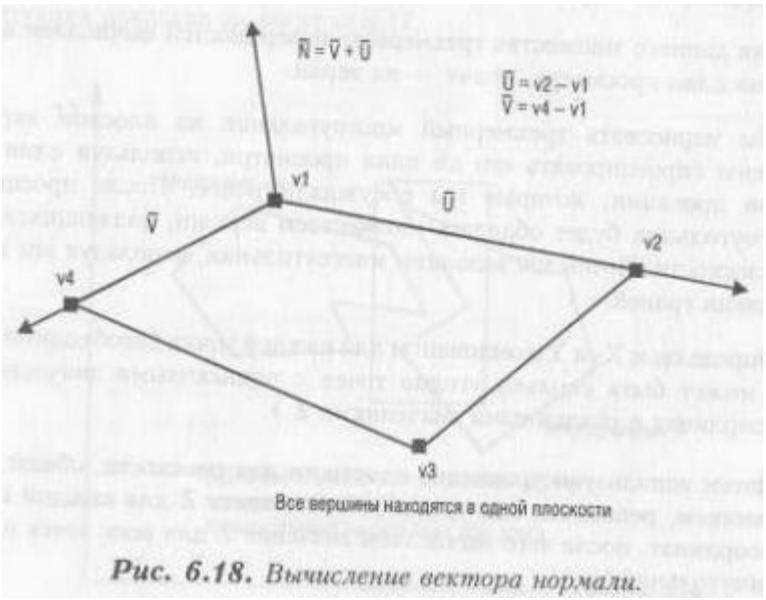


Рис. 6.18. *Вычисление вектора нормали.*

Имея вектор нормали к многоугольнику, уравнение плоскости находит Z-компонент для любой точки (x,y). При этом заданы: искомая точка (x,y) и вектор нормали к многоугольнику <Nx,Ny,Nz>:

$$Z = \frac{Nz}{1 - Nx * X - Ny * Y}$$

Соотношение пространства образов и пространства объектов

Алгоритм Z-буфера хорошо работает и может быть легко реализован. Единственная проблема состоит в том, что он работает на уровне пикселей и является алгоритмом обработки образа. Это значит, что он не рассматривает геометрических свойств объекта. Это требует наличия некоторого гибридного алгоритма для использования в специальных случаях. Такой алгоритм должен учитывать геометрические свойства объекта перед простым удалением невидимых поверхностей. Теперь давайте поговорим о том, как придать поверхности наших трехмерных объектов большую реалистичность.

Трассировка луней

Трассировка лучей - это метод, применяемый для создания реалистичных образов на

компьютере, используя полные модели трехмерного мира. Трассировка лучей решает множество проблем. Этот алгоритм может выполнять следующие действия:

- Удаление невидимых поверхностей;
- Перемещение;
- Отражение;
- Рассеяние;
- Окружающее освещение;
- Точечное освещение;
- Наложение теней.

Изначально этот алгоритм разрабатывался для решения проблемы удаления невидимых поверхностей. Трассировка лучей создает образ, исходя из тех же законов, что и наше зрение. На рисунке 6.19 изображено некоторое пространство, которое может быть просчитано с помощью алгоритма трассировки лучей. Вы видите несколько объектов: источник света, наблюдателя и план наблюдения.

Чтобы воспользоваться трассировкой лучей для создания натуральных образов, нам придется использовать миллиарды световых лучей из источника света, и затем рассматривать каждый из них, надеясь, что он попадет в план наблюдения и примет участие в создании образа. Возникает вопрос - а зачем трассировать каждый возможный луч? На самом деле, нас интересуют только те лучи, которые достигают плана просмотра.

Запомнив это, давайте попробуем трассировать лучи в обратном направлении. Проследим движение лучей для каждого из пикселей на экране, а затем посмотрим, где эти лучи пересекаются с планом просмотра. Отметив пересечение, мы останавливаемся и окрашиваем соответствующий пиксель в нужный цвет. Это называется первичной трассировкой лучей.

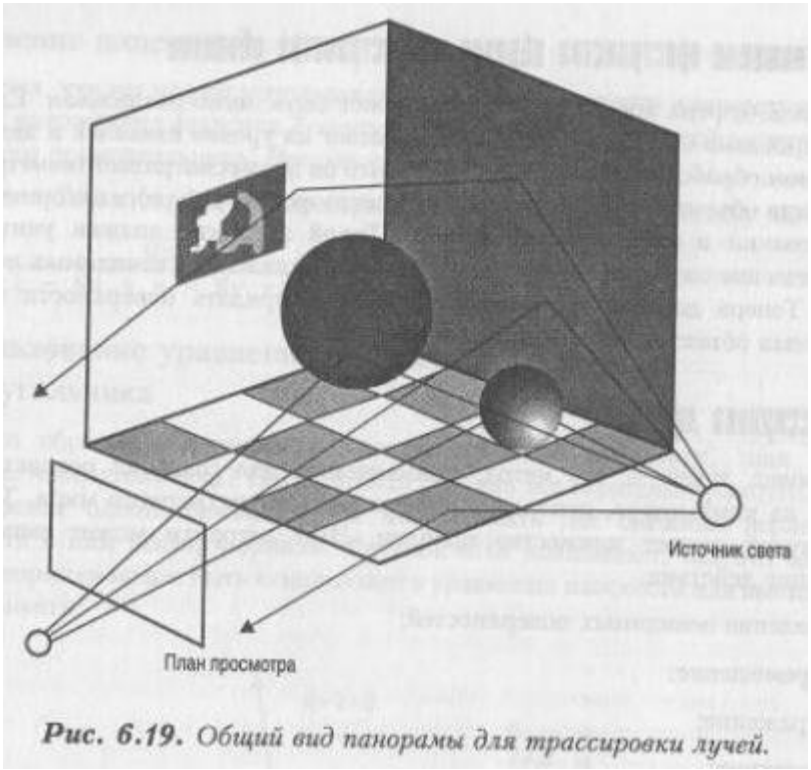


Рис. 6.19. *Общий вид панорамы для трассировки лучей.*

Данная техника позволяет создавать трехмерные образы, но при этом не видны такие эффекты, как тени, рефракция и рефлексия. Чтобы воссоздать перечисленные эффекты, мы должны принять в рассмотрение специальные вторичные лучи, которые исходят из точек пересечения. Это все делается рекурсивно до достижения некоторого уровня детализации. Затем полученные по всем лучам результаты складываются и соответствующему пикселю присваивается вычисленный цвет.

Трассировка лучей - это один из наиболее насыщенных вычислениями методов расчета трехмерных изображений, но зато и результаты получаются впечатляющими. Есть только одна проблема: для решения этой задачи в реальном времени не хватает мощности даже самого быстродействующего компьютера. Потому нам придется учесть данное обстоятельство и применить идею трассировки лучей для создания другого метода. Он будет более ограничен, но позволит нормально работать с трехмерными мирами на обычном ПК. Исходя из этого, мы попробуем реализовать упрощенный вариант трассировки лучей, используя только первичные лучи для генерации изображения. С последующими оптимизациями возможно достижение достаточно высокой производительности. Если вам интересно узнать, как это можно сделать, то стоит продолжить чтение нашей книги.

Отсечение лучей

Для создания реалистичного трехмерного изображения в играх используется техника, называемая отсечением лучей. Применяя эту технику, надо придерживаться некоторых правил, связанных с тем, что алгоритм отсечения лучей — это, в сущности, упрощение алгоритма трассировки, в котором все же осталось множество вычислительных проблем. Поэтому данный алгоритм применим только для наиболее простой геометрии создаваемых миров. В общем случае, отсечение лучей не будет работать, если вы решите сделать трехмерный имитатор полетов или попытаетесь смоделировать реальное пространство. Но в играх, где действие происходит в мире, нарисованном с помощью перпендикулярных линий, этот алгоритм работает изумительно. Для создания DOOM-образных миров применяется несколько иная техника, но и она базируется на этом же алгоритме.

Я написал маленькую демонстрационную программку, в основе которой лежит алгоритм отсечения лучей-. Она позволит вам «погулять» по трехмерному миру, состоящему из кубов. Весь этот мир на самом деле является двухмерной матрицей, считываемой из обычного ASCII-файла.

Я решил использовать графическую библиотеку Microsoft, поскольку сейчас меня не волнует вопрос быстродействия. Я решил, что вам надо иметь перед глазами двух- и трехмерные картинки, поэтому использовал режим высокого разрешения и соответствующие библиотеки. Это дает лучшие ощущения для понимания механизма процесса.

Следующие страницы будут насыщены деталями и техническими подробностями. Отсечение лучей теоретически просто, но практическая реализация весьма сложна. Это связано с тем, что приходится принимать во внимание кучу мелких деталей. Эти детали очень важны. Я покажу вам очень простую программу отсечения лучей, но основной задачей будет понять принцип ее работы.

Поскольку здесь я использовал библиотеки поддержки математических операций с плавающей запятой, то программа работает довольно медленно. Когда вы ее оттранслируете и запустите, то увидите три окна вывода. Они показаны на рисунке 6.20. Изображение в левом углу экрана представляет собой плоскую карту Трехмерного мира, который является матрицей размером 64x64x64.

Полная трехмерная модель из этой матрицы создается посредством отсечения лучей. Как это получается, изображено в правой части экрана. Для перемещения используйте цифровую панель клавиатуры. Чтобы выйти из Программы нажмите клавишу Q. В процессе работы с программой у вас должно, Появиться представление о строении трехмерного образа: он построен из вертикальных полос. Эти полосы образуются в процессе поворота луча в точке просмотра на определенный угол.

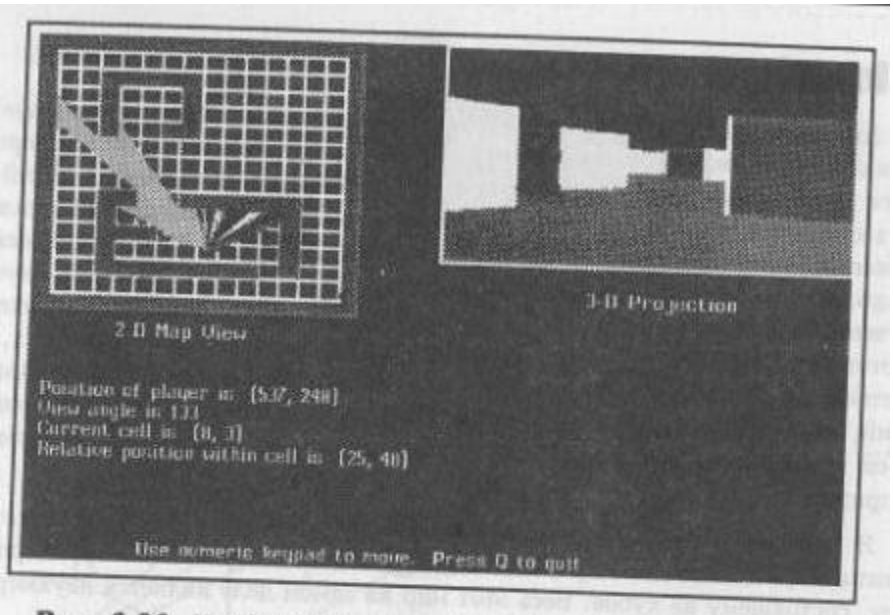


Рис. 6.20. *Изображение экрана вывода программы RAY.EXE.*

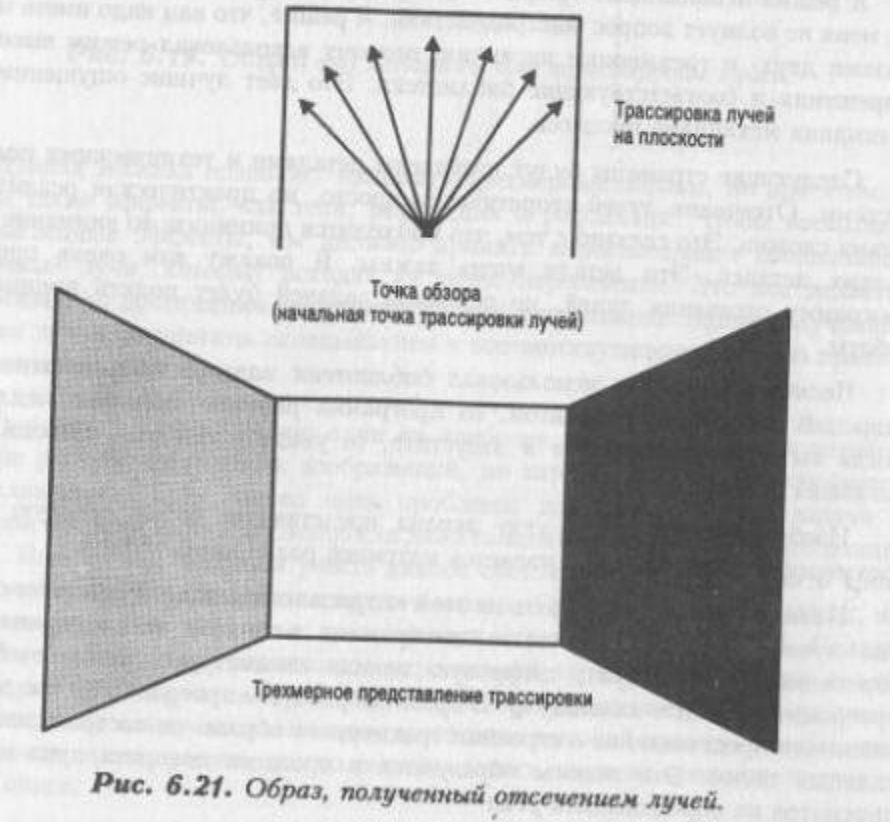


Рис. 6.21. *Образ, полученный отсечением лучей.*

Идею алгоритма отсечения лучей можно представить так: вообразите, что вы стоите в пустой комнате и смотрите вперед. Все, что вы наблюдаете, это стены впереди и по обе стороны от вас. Образ, который вы видите, складывается из лучей, отразившихся от стен и попавших в ваши глаза. Но при отсечении лучей происходит не отражение от стен, а просто отсечение лишних лучей. На рисунке 6.21 показан образ, полученный таким методом.

Как и в системах лазерного сканирования, мы также как бы сканируем область вокруг нас и строим образ на основе полученных результатов. То, что мы в итоге получим, будет зависеть от поля просмотра. Это поле является той «порцией» информации, которую мы можем увидеть за один раз. Если мы способны обозреть пространство под углом 45° вправо и влево по отношению к направлению просмотра (рис. 6.22), то наше поле просмотра составит 90°.

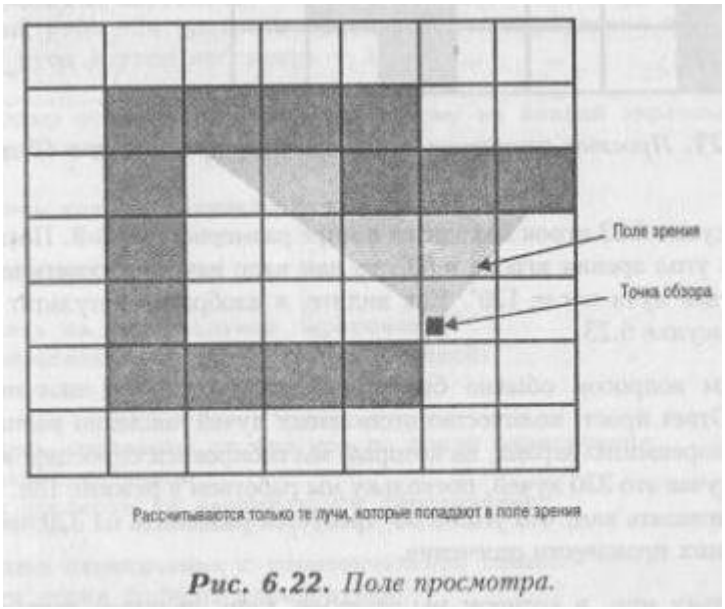


Рис. 6.22. *Поле просмотра.*

Вообще, поле просмотра - это одно из ключевых понятий в технологии отсечения лучей. Поэтому мы должны определить, какое поле просмотра нам необходимо. Большинство животных имеет очень большое поле просмотра - 90 и более градусов. Правда, для нашего случая я выбрал его равным 60°. Просто мне нравится то, что получается при этом на экране. Вы сами можете задать любой другой угол, но постарайтесь, чтобы он попадал в диапазон между 60 и 90 градусами.

Теперь мы знаем, что нам надо отсечь все лучи, которые не попадают в наше поле просмотра. Потом, нам надо выяснить точки пересечения этих лучей со стенами и использовать информацию о каждом пересечении для построения трехмерного образа. Для примера посмотрим на рисунок 6.23.

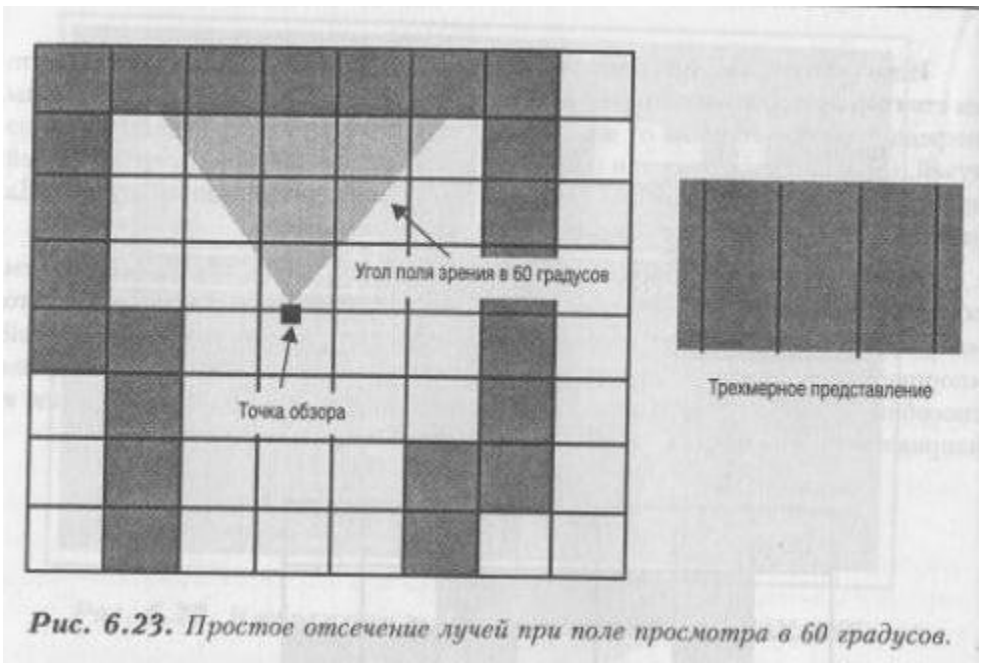


Рис. 6.23. *Простое отсечение лучей при поле просмотра в 60 градусов.*

На рисунке 6.23 игрок находится в мире размерностью 8x8. Поскольку мы установили угол зрения игрока в 60°, то нам надо начать отсекать все лучи до угла 30° и все лучи после 120°. Как видите, я изобразил результат отсечения лучей на рисунке 6.23.

Первым вопросом обычно бывает: «А сколько лучей нам необходимо отсечь?». Ответ прост; количество отсекаемых лучей численно равно горизонтальному разрешению экрана, на который мы собираемся спроецировать образ. В нашем случае это 320 лучей, поскольку мы работаем в режиме 13h. Интуиция должна подсказать вам, что угол в 60° требуется разделить на 320 частей и для каждой из них произвести отсечение.

Поскольку мир, в котором мы отсекаем лучи, является двухмерным, то задача вычисления пересечений становится довольно простой. Более того, наш мир имеет регулярную структуру. Это значит, что количество вычислений резко уменьшается. Впоследствии я покажу множество маленьких хитростей, которые заставят работать этот алгоритм с фантастической скоростью (мы это сделаем чуть позже).

Мы имеем набор лучей, распределенных в диапазоне от -30 до +30 градусов к лучу зрения. Как мы уже говорили, поле просмотра у нас равно 60°. Давайте смоделируем на экране поле просмотра. Для этого нам нужно:

1. Отсечь 320 лучей (один для каждого вертикального столбца на экране) и вычислить пересечение каждого луча с блоками, из которых состоит наша двухмерная карта мира;

2. Используя эту информацию, вычислим дистанцию между игроком и точкой пересечения;
3. Затем используем эту дистанцию для масштабирования вертикальной полосы. Горизонтальная позиция при этом соответствует координате текущего луча (0..319).

Алгоритм 6.1 показывает последовательность действий при отсечении лучей.

Алгоритм 6.1. Алгоритм отсечения лучей.

```
// Пусть игрок находится в позиции (xp,yp) и его взгляд
// Направлен под углом view_angle
// Инициализируем переменные
// Начальный угол -30 градусов относительно направления взгляда
стартовый угол = угол просмотра - 30;
// необходимо отсечь 320 лучей, по одному на каждый экранный столбец
for (ray=0; ray<320; ray++)
{
вычислить наклон текущего луча
while (луч не отсечен)
{
// проверить на вертикальное пересечение
if (не пересекается с вертикальной стеной)
if (луч отсек блок по вертикали)
{
вычислить дистанцию от (xp,yp) до точки пересечения
сохранить дистанцию
} //конец проверки на вертикальное пересечение
if (не было пересечения с горизонтальной стеной)
if (луч отсек блок по горизонтали)
{
вычислить дистанцию от (xp,yp) до точки пересечения
сохранить дистанцию
} // конец проверки по горизонтали
} // конец цикла while
if (горизонтальное пересечение ближе вертикального)
{
вычислить масштаб по горизонтали
нарисовать полосу изображения
}
// конец оператора if
else // вертикальное пересечение ближе горизонтального
{
вычислить масштаб по вертикали нарисовать полосу изображения
} // конец оператора else
} // конец
```

Конечно, мы опустили множество деталей, но при этом четко формализовали основную идею алгоритма.

Единственный вопрос, который может смутить: «А почему это всегда работает?». Просто мы смоделировали процесс прорисовки образа частицами света. Правда, проделали мы это в обратную сторону, но главное — такой метод работает. Он удаляет невидимые поверхности, создает перспективу и содержит всю необходимую информацию для создания теней, освещения и текстур. Именно поэтому алгоритм отсечения лучей является очень мощным средством для программиста. Мы можем создавать в играх окружение, которое было бы невозможно получить, используя стандартную технику работы с многоугольниками.

Вы можете сказать, что мир, построенный из одинаковых блоков, выглядит весьма скучно. Да, это так, но если вы добавите тени и разукрасите стены всевозможной фактурой, все сказочно преобразится. Вы сможете создать восхитительное окружение для ваших игр. Впоследствии вы сможете уменьшить размер блоков для создания более сложных сцен. Кроме того, вы научитесь изображать поверхности с углом наклона в 45°. Все в ваших руках.

Вас может остановить только быстродействие, поэтому надо постоянно искать пути для сокращения времени выполнения на всех этапах обработки, изображений. Тем более, что DOOM уже доказал: для ПК нет ничего невозможного.

Теперь, когда мы узнали основы метода отсечения лучей, давайте погрузимся в детали его реализации и всей той математики, которая для этого необходима. Это один из наиболее важных пунктов настоящей главы, да, пожалуй, и всей книги. Я прочитал множество книг, описывающих различные алгоритмы, но не нашел, как и где их применить. Все это приходилось выяснять самому, методом проб и ошибок. Поэтому я решил рассказать вам про все тонкости и детали, которые мне известны.

Правда, надо помнить вот о чем. Я написал программу отсечения лучей так, чтобы она легко читалась и была понятна. Это не значит, что с точки зрения скорости она идеальна. Нет, скорее

наоборот. Но зато вы без особых усилий поймете принципы работы алгоритмов.

Математические основы отсечения лучей

Отсечение лучей, с точки зрения теории, вещь простая. Отсекается определенное количество лучей и вычисляются координаты точек, где они пересекаются с каждой вертикальной или горизонтальной линией. Все это выглядит довольно просто. Проблема состоит в том, чтобы выполнять это быстро и эффективно.

В этом разделе мы поговорим о проблеме поиска пересечений, поскольку именно в этой части программа тратит больше всего времени. Есть семь основных моментов, которые мы должны проанализировать и понять:

- рисование лучей;
- вычисление координат первого пересечения;
- вычисление координат следующего пересечения;
- вычисление.расстояния;
- масштабирование;
- уменьшение проекционного искажения;
- рисование полос.

Рисование лучей

Очевидно, что лучи, которые мы отсекаем, на самом деле представляют собой линии. Они начинаются в точке зрения игрока, совпадающей с его позицией на двухмерной карте. Мы решили иметь поле просмотра равным 60°. Таким образом, нам нужно составить таблицу соответствия для всех возможных лучей, которые можно отсечь с любого угла просмотра. Эта таблица должна содержать значения наклонов всех возможных лучей по отношению к плану просмотра. Исходя из значения наклона, мы сможем произвести отсечение луча из точки наблюдения игрока.

Формула 6.1. Подсчет количества элементов в таблице значений наклонов.

Перед нами стоит вопрос — сколько элементов должно быть в таблице наклонов и как эти наклоны рассчитать? Для составления таблицы требуется знать, сколько в ней будет элементов. Когда игрок смотрит на мир, построенный отсечением лучей, то 320 лучей (количество горизонтальных точек экрана) вместе составят дугу в 60°. Таким образом, мы должны иметь таблицу с 1920 элементами или значениями наклонов. Это вычисляется по следующей формуле:

```
размер таблицы = ширина_экрана x 360 / поле_просмотра
```

В нашем случае ширина экрана составляет 320 пикселей, а поле_просмотра - 60 градусов, поэтому результат будет равен 320x360/60 = 1920.

Определение значения наклона

Теперь нам надо вычислить каждый из элементов таблицы наклонов. Это должны быть действительные значения наклонов всех возможных линий, отсекаемых от точки наблюдения игрока. Поскольку мы разбили окружность на 1920 секторов, то каждый из них будет по 360/1920 = 0.1875 градусов. Таким образом, мы нашли способ вычисления наклона для всех линий окружности с шагом в 0.1875 градуса. Все это выглядит довольно сложным для вычисления. К счастью, проблему поможет решить функция tg (). Если вы забыли определение тангенса, то напомним, что для прямоугольного треугольника он равен отношению синуса к косинусу угла:

```
                                        SinQ
противоположная_сторона
TgQ      =      --------                =
```

Если это так, то

M = tg q

Создав таблицу значений наклонов из 1920 элементов, где каждый наклон равен tg q, все остальные расчеты значительно упрощаются. Мы используем эти наклоны для построения лучей. Но есть несколько проблем:

- Только в первом квадранте значения тангенса будут корректны. В других квадрантах он может быть как отрицательным, так и положительным, но вы не сможете узнать знак наклона, поскольку сама функция является частным. Таким образом, при программировании надо на это обратить внимание и выполнять вычисления в условном операторе;
- Функция tg() имеет вертикальные асимптоты при углах в 90 и 270 градусов. Поэтому надо быть внимательным, чтобы избежать в этих точках деления на ноль или ошибок с плавающей запятой.

Вычисление линий для генерации

Поскольку мы сумели успешно заполнить таблицу значения наклонов, то теперь готовы ею воспользоваться для вычисления линий.

1. Находим позицию игрока для текущего отсечения лучей или рендеринга.
2. Вычисляем первое пересечение для каждого луча, который отсекается пересечением.

Помните, что игрок находится на плоской карте, которая используется для создания трехмерного образа. В нашем случае площадь мира составляет 16x16 ячеек, и каждая ячейка имеет размер 64x64 пикселя. Таким образом, мир имеет 1024x1024 виртуальных единиц измерения. Вне зависимости от позиции, игрок будет занимать несколько ячеек игрового пространства, поскольку «размер» самого игрока равен размеру ячейки. Эта позиция вычисляется простым делением глобальной позиции игрока, которая меняется от 0 до 1023 для X и Y, на 64,или

```
ячейка_x = x_размерность / 64
ячейка_y = y_размерность / 64
```

где x_размерность и y_размерность находится в пределах от 0 до 1023.

Когда координаты текущей игровой ячейки будут вычислены, мы сможем найти ее пересечение с текущим лучом. Давайте рассмотрим математику для выполнения этих действий.

Вычисление точки первого пересечения

Существует множество способов описания прямой: от параметрического до функционального. Мы будем рассчитывать нашу прямую с помощью функции, описывающей линию.

формула 6.2. Вычисление первой Y-координаты пересечения.

$$\frac{(Yi - Yp)}{M} =$$

где (Xi,Yi) - точки пересечения линии и (Xp,Yp) — позиция игрока. После некоторых алгебраических преобразований получаем:

$$Yi = M * (Xi - Xp) +$$

Формула 6.3. Вычисление первой X-координаты пересечения.

$$Xi = M^{-1} * (Yi - Yp) +$$

Для нахождения первого пересечения текущего луча нам надо выполнить только два умножения и сложение. Не так плохо. Мы обязательно это оптимизируем, но сделаем это позже. Переменная M — это величина наклона, которую мы уже вычислили и занесли в таблицу наклонов.

Единственное, что может нас расстроить в этих преобразованиях, это их рекурсивность: каждое преобразование требует предварительного вычисления Другого. Это несколько затруднительно. Но вы можете заметить, что Xi в первом равенстве на самом деле означает «первая граничная вертикальная линия», а Yi во втором выражении означает «первая граничная горизонтальная линия». В этом и заключена изюминка метода отсечения лучей для квадратных матриц. Далее, мы знаем, что луч пересекается с каждой из ячеек в вертикальном и горизонтальном столбце. Рисунок 6.24 поясняет это, Если мы однажды вычислим первое пересечение, то сможем найти и все остальные пересечения с лучом, а также конец его траектории.

Прежде чем рассказать вам о поиске остальных пересечений, я хочу отметить, что каждый: луч может иметь пересечения как с вертикальной, так и с горизонтальной асимптотами, и мы должны вычислять эти пересечения.

Некоторые, программисты любят это делать за два прохода: в первом вычисляются все возможные горизонтальные пересечения, а во втором — все вертикальные пересечения. Рисунок 6.25 демонстрирует эту идею.

Это нормально, но я предпочитаю делать это одновременно. Время выполнения обоих методов одинаково, а остальное - дело вкуса. Есть еще одна деталь в работе с пересечениями, для каждого из них (включая самое первое) мы проверяем наличие непрозрачного объекта. Мы должны посмотреть во все стороны от пересечения и попробовать его заметить. Направление просмотра зависит ог нашего положения. Если мы находим объект, то останавливаемся и вычисляем дистанцию до пересечения (мы научимся это делать позже). Если мы не обнаруживаем пересечений, то просто продолжаем вычисления до тех пор, пока луч не упрется в какой-либо объект или не выйдет за границы нашего игрового мира.

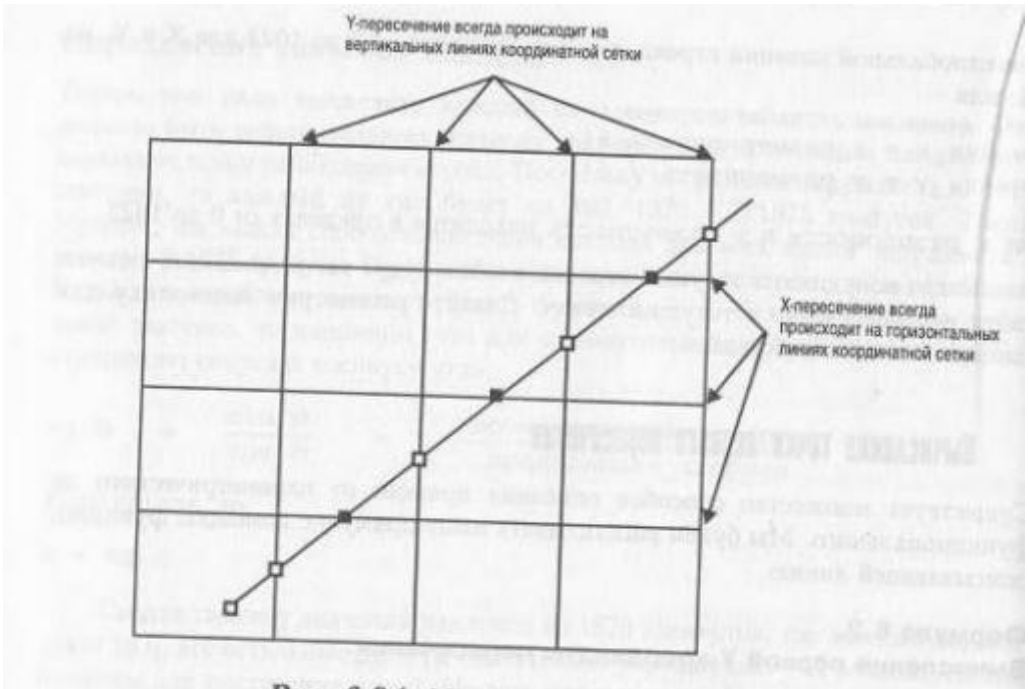


Рис. 6.24. Механика пересечения линий.

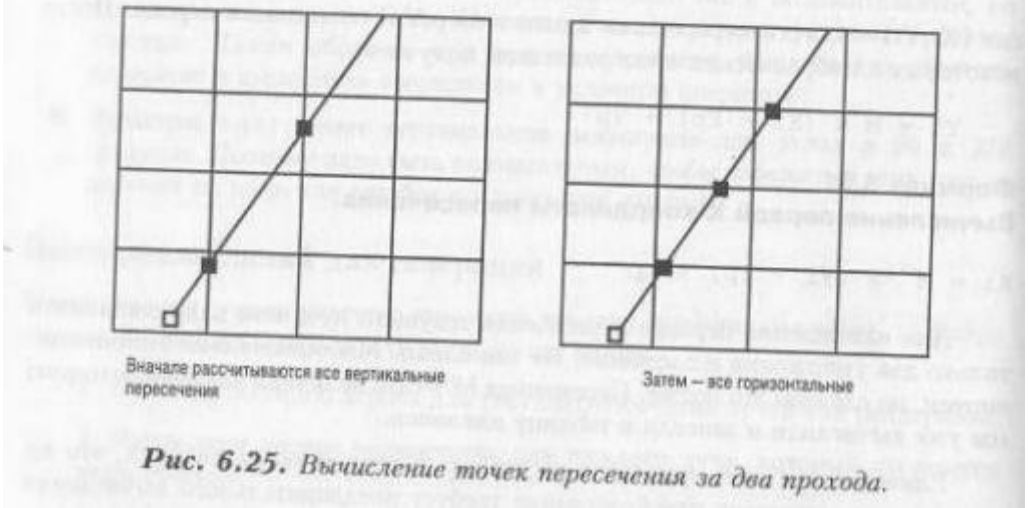


Рис. 6.25. Вычисление точек пересечения за два прохода.

Вычисление оставшихся пересечений

После того как мы нашли первое пересечение для вертикальной и горизонтальной стены, можно найти следующую точку возможного пересечения — достаточно прибавить константы к текущей точке пересечения. Например, если координата Yi пересечения равна 100, то следующая точка Yi может быть рассчитана простым прибавлением числа 100.

В нашем случае каждая ячейка имеет размеры 64x64, и чтобы найти следующее пересечение мы используем следующие формулы:

Формула 6.4. Вычисление X-координаты следующего возможного пересечения.

Следующее Xi = Xi + M x (ширина ячейки)

Формула 6.5. Вычисление Y-координаты следующего возможного пересечения.

Следующее Yi = Yi + M x (высота ячейки)

где высота и ширина ячейки равна 64.

Это не так плохо: вычисление следующего возможного пересечения требует двух умножений и двух сложений. Помните, отсечение лучей должно выполняться максимально быстро, а иначе в нем нет смысла.

После вычисления следующей возможной точки мы проверяем, есть ли там что-то на самом деле. Если есть, то мы вычисляем дистанцию до этого пересечения. Дистанция используется позже, для трехмерного рендеринга. Если же пересечения нет, то мы продолжаем отсечение.

Вычисление расстояния

Теперь, когда мы нашли пересечение с объектом, встает вопрос о том, как вычислить расстояние до него. Поскольку мы нашли точку пересечения, то можем найти и расстояние от этой точки до позиции игрока. Это расстояние нам нужно для двух вещей:

- Чтобы узнать тип пересечения (вертикальное или горизонтальное);
- Чтобы выяснить масштаб стен.

Формула 6.6. Теорема Пифагора.

Мы можем использовать для этого теорему Пифагора;

$H = sqrt (x^2 + y^2)$

Это прекрасная формула, но ужасно сложная для ПК, поскольку вычисление квадратного корня — процесс весьма медленный и долгий. Мы, конечно, можем использовать табличное представление функции квадратного корня, но и в этом нет ничего хорошего. Есть другой, более простой способ найти удаление точки пересечения от игрока, если вычислить координаты точек и найти sin © и cos ©.

Давайте посмотрим на рисунок 6.26.

Поскольку мы уже вычислили точки вертикального и горизонтального пересечения, мы можем воспользоваться известным правилом, чтобы найти синус и косинус угла наклона:

- Угол наклона луча (он у нас есть);
- Длина гипотенузы (мы ее хотим найти);
- Длина сторон треугольника (который мы имеем).

Назвав переменные так же, как они обозначены на рисунке 6.26, напишем формулы для вычисления длины- гипотенузы (или, что то же самое - искомого расстояния).

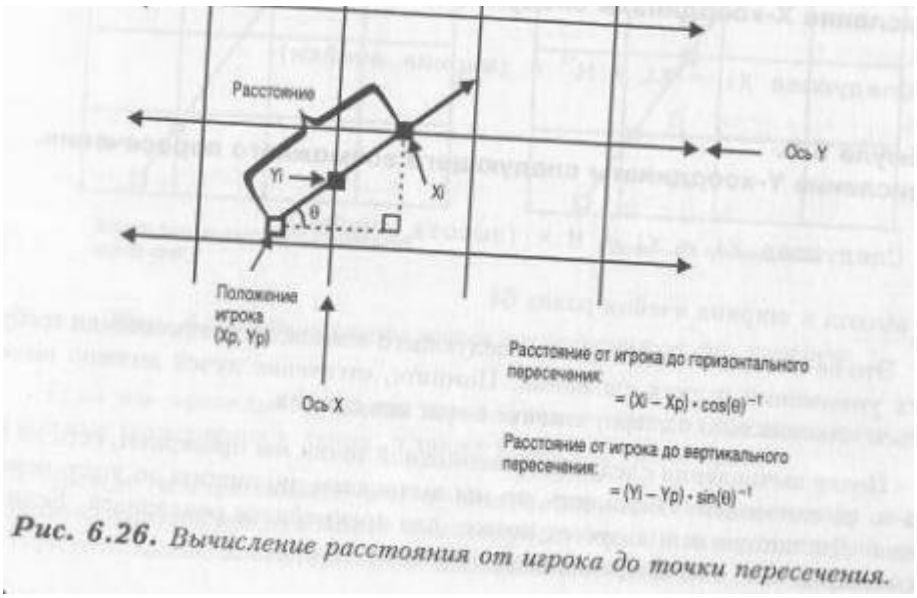


Рис. 6.26. *Вычисление расстояния от игрока до точки пересечения.*

Формула 6.7. Вычисление расстояния до точки X-пересечения.

расстояние = (Xi - Xp) x $\cos^{-1}$ A

Формула 6.8. Вычисление расстояния до точки Y-пересечения.

расстояние = (Yi - Yp) x $\sin^{-1}$ A

где A - это угол луча, который рассматривается в настоящее время. В программе это просто индекс от 0 до 1920 для таблицы вычисленных значений $\sin^{-1}$ и $\cos^{-1}$.

Это может показаться громоздким, но если это правильно написать и оптимизировать, то процесс будет занимать несколько миллисекунд. После того как расстояние будет вычислено, встает вопрос, что ближе, а что дальше на экране.

Вычисление масштаба

Теперь поговорим о масштабах. В виртуальном мире компьютера понятие масштаба превращается в понятие относительного масштаба. Таким образом, об этом можно говорить только в сравнении с чем-либо другим.

Для вычисления масштаба мы применяем сравнение между X- и Y-пересечениями. Масштаб или высота вычисляется на основе ближайшего пересечения. Когда мы вычисляем масштаб битовой карты, мы можем использовать само расстояние:

```
масштаб = расстояние до пересечения
```

Это не всегда срабатывает: приближаясь, объекты становятся больше, но не уменьшаются при удалении. Тогда можно применить такую формулу:

```
масштаб = 1 / расстояние до пересечения
```

Это работает. Но результат надо умножить на некоторый коэффициент расстояния (или правдоподобия — как вам больше нравится).

```
масштаб = K / расстояние до_пересечения
```

где K стоит подобрать самостоятельно.

Уменьшение проекционных искажений

Проблема, о которой нам надо поговорить, заключается в проекционном искажении. Как вы знаете, для реализации отсечения лучей мы нарушили правило и использовали одновременно полярные и декартовы системы координат. Это привело к эффекту «рыбьего глаза» (то же самое возникает,когда вы смотрите сквозь сильную линзу).

Это сферическое искажение возникает вследствие использования нами при трассировке лучей радиального метода. Мы рассчитываем все лучи, выходящие из одной точки (позиции игрока). Сферические искажения возникают потому, что все объекты, с которыми пересекаются лучи, определены в «прямоугольном» пространстве. Расчет же лучей проводится в «сферическом» или «полярном» пространстве. Пример такого пространства представлен на рисунке 6.27.

Рис. 6.27. Полярное пространство трассировки лучей.

Теперь посмотрим на рисунок 6.28. Мы увидим, что наблюдает игрок, когда смотрит прямо

на стену. Он видит прямоугольник. Но так как расстояния до точек пересечения различны, изображение получается искаженным. Рисунок 6.29 показывает два результата отсечения лучей. Первый построен с учетом компенсационных искажений, а второй — без их учета. Все это очень интересно, но как это реализовать? Ответ прост: нужно умножить функцию масштаба на инверсную функцию. Синусоидальное искажение может быть компенсировано умножением масштаба на $\cos^{-1}$ текущего угла по отношению к полю наблюдателя (60 градусов). То есть мы должны умножить каждое значение угла от -30 до +30 на $\cos^{-1}$ того же угла. Это исключит искажение.

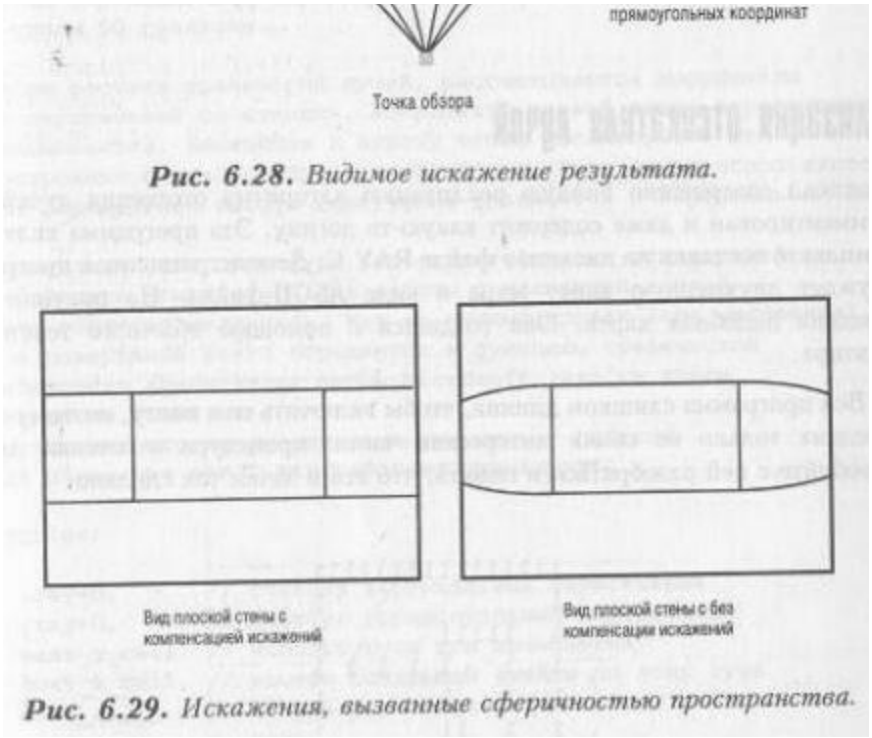


Рис. 6.28. *Видимое искажение результата.*

Рис. 6.29. *Искажения, вызванные сферичностью пространства.*

Отрисовка фрагментов стен

Наконец, мы должны нарисовать фрагмент битовой карты, который представляет одну вертикальную полосу окончательно сформированного экрана. Как я уже говорил, размер вертикальных фрагментов рассчитывается относительно средней линии, делящей экран на две горизонтальные плоскости. Таким образом, после того как мы рассчитали окончательный масштаб изображения, верхняя и нижняя границы фрагмента могут быть определены по формуле:

```
top    = 100 - scale/2 // верхняя граница
bottom = top + scale   // нижняя граница
```

где scale — окончательные вертикальны

```
1111111111111111
1               1
1   11111       1
1   1   1 1 1   1
1   1   1       1
1   1  11       1
1               1
1               1
1               1
1               1
1   11  111111  1
1   1         1 1
1   111       1 1
1   1         1 1
1   11111111  1 1
1               1
1111111111111111
```

Реализация отсектеля лучей

Я написал совершенно полну оритма отсечения лучей. Он закомментирован и даже содержит как ограмма включена в комплекте поставки на дискете в файле RAY. C. Дем мма загружает двухмерную карту мира в виде ASCII-файла. На рисунке 6.3 карта. Она создается с помощью обычного текстового редактора.

Вся программа слишком длинна, книгу, поэтому здесь приведена только ее самая интересная часть: проце Попробуйте с ней разобраться и понять, что это и зачем так сделано.

Листинг 6.4. Процедура отсечения лучей (RAYLIST.C)

```
void Ray_Caster(long x, long y,long view_angle)
{
// эта функция выполняет расчет 320 лучей и строит игровой
// экран на основе их пересечений со стенами. Расчет производится
// таким образом, что все лучи отображаются на поле просмотра
// с углом 60 градусов
// после расчета траекторий лучей, рассчитываются координаты
// их пересечений со стенами. Координаты первой точки пересечения
// запоминаются. Ближайшие к игроку точки используются для
// построения битового образа изображения. Расстояния используются
// для определения высоты текстурных фрагментов и вертикальных линий
// Примечание: эта процедура использует функции стандартной
// библиотеки компилятора для работы с плавающей точкой
// (это работает медленно). Код не оптимизирован (еще медленнее),
// и в довершение всего обращается к функциям графической
// библиотеки компилятора фирмы Microsoft (что уж вовсе
// никуда не годится!) // Однако все это имеет определенную цель - легкость
// для понимания того, как работает процедура
int rcolor;
long xray=0,      // счетчик вертикальных пересечений
yray=0,      // счетчик горизонтальных пересечений
next y cell, // используются для вычисления
next_x cell, // номера следующей ячейки по ходу луча
cell_x,      // координаты текущей ячейки
ce11_y,      // луча
x_bound,     // следующие вертикальная
y_bound,     // и горизонтальная точки пересечения
xb_save,     // в этих переменных запоминаются
yb_save,     // координаты точек пересечения
x_delta,     // эти переменные показывают, на сколько
y_delta,     // надо сместиться для перехода к следующей ячейке
ray,         // текущий луч для отсечения
casting=2,   // показывает компоненты X и Y луча
x_hit_type,  // координаты блока, с которым пересекся луч
y_hit_type,  // используются при отрисовке
top,         // верхняя и нижняя координаты области,
bottom;      // отрисовываемой как стена (с помощью текстуры)
```

```
float xi,         // используется для определения x- и y-пересечений
yi,
xi_save,    // используется для сохранения
// точек пересечения X и У
yi_save,
dist_x,     // расстояние до x- и y-пересечениЙ
dist_y,     //  от точки просмотра scale;
// масштаб
//СЕКЦИЯ   1 ////////////////////////////////////
// инициализация
// вычисляет начальный угол от игрока. Поле просмотра 60 градусов.
// Таким образом, рассматриваем только половину - 30 градусов
if ( (view_angle-=ANGLE_360) < 0)

// разворачиваем вектор направления взгляда
view_angle=ANGLE_360 + view_angle;
} // конец оператора if
// выбираем цвет для луча
rcolor=1 + rand()%14;

//СЕКЦИЯ   2 ////////////////////////////////////
// цикл для всех 320 лучей
for (ray=0; ray<320; ray++)
// вычислить первое x-пересечение
if (view_angle >= ANGLE_0 && view_angle < ANGLE_180)
{
// вычислить первую линию, которая пересекается с лучом.
// Примечание: эта линия должна быть выше (впереди
// на игровом поле) игрока.
y_bound = CELL_Y_SIZE + CELL_Y_Sf2E * (y / CELL_Y_SI2E);
// вычислить смещение для перехода к следующей
// горизонтальной линии
y_delta = CELL_Y_SIZE;   // размер ячейки по вертикали (ред.)
// основываясь на первой возможной горизонтальной линии отсечения,
// вычислить X-пересечение и начать расчет
xi = inv_tan_table[view_angle] * (y_bound - y) + x;
// установить смещение
next_y_cell = 0;
} // конец обработки верхней половины плана
else
{ // вычислить первую горизонтальную линию, которая может
// пересекаться с лучом. Это будет позади игрока
y_bound = CELL_Y_SI2E * (y / CELL_Y_SIZE);
// вычислить смещение для следующей горизонтальной линии
y_delta = -CELL_Y_SIZE;
// основываясь на первой возможной горизонтальной линии отсечения,
// вычислить X-пересечение и начать расчет
xi = inv_tan_table[view_angle] * (y_bound - y) + x;
next_y_cell = -1;
} // конец обработки нижней половины плана
//СЕКЦИЯ   3 ////////////////////////////////////
// вычислить первое x-пересечение
if (view_angle < ANGLE_90 || view_angle >= ANGLE_270) {
// вычислить первую вертикальную линию, которая будет
// пересекаться с лучом. Она должна быть справа от игрока
x_bound = CELL_X_SIZE + CELL_X_SIZE * (x / CELL_X__SIZE);
// вычислить смещение
x_delta = CELL_X_SIZE;
// основываясь на первой возможной вертикальной линии отсечения,
// вычислить Y-пересечение и начать расчет
yi = tan_table[view_angle] * (x_bound - x) + y;
next_x_cell = 0;
} // конец обработки правой половины плана
else
{
// вычисляем первую вертикальную линию, которая может быть
// пересечена лучом. Она должна быть слева от игрока
```

```
x_bound = CELL_X_SIZE * (x / CELL_X_SIZE);
// вычислить расстояние до следующей вертикальной линии
x_delta = -CELL_X_SIZE;
// основываясь на первой возможной вертикальной линии отсечения,
// вычислить Y-пересечение
yi = tan_table[view_angle] * (x__bound - x) + y;
next_x_cell = -1;
}
// начать отсечение
casting      =2;   // два луча для одновременного отсечения
xray = yray = 0;   // сбросить флаги пересечения
//СЕКЦИЯ   4 ////////////////////////////////////////
while(casting)
{
// продолжить отсечение лучей
if (xray!=INTERSECTION_FOUND)
{
// тест на совпадение луча с асимптотой
if (fabs (y_step[view_angle])==0)
xray = INTERSECTION_FOUND;
casting--;
dist_x = 1e+8;
} // конец проверки на совпадение с асимптотой
// вычислить текущую позицию карты для проверки
cell_x = ( (x_bound+next_x_cell) / CELL_X_SIZE);
cell_y = (long)(yi / CELL_Y_SIZE) ;
// проверить, есть ли в этом месте блок
if ((x_hit_type = world[(WORLD_ROWS-1) - cell_y][cell_x])!=0)
{
// вычислить расстояние
dist_x = (yi - y) * inv_sin__table[view angle];
yi_save == yi;
xb_save = x_bound;
// закончить x-отсечение
xray = INTERSECTION_FOUND;
casting--;
} // конец проверки попадания луча на стену блока
else
{
// вычислить следующее Y-пересечение
yi += y_step[view_angle];
} //конец оператора else
} // конец проверки на отсечение по оси X
//СЕКЦИЯ   5 ////////////////////////////////////////
if (yray!=INTERSECTION_FOUND)
{
// тест на попадание луча на асимптоту
if (fabs(x_step[view_angle])==0)
{
yray = INTERSECTION_FOUND;
casting--;
dist_y=1e+8;
}

// вычислить позицию карты
ceil_x = (long)(xi / CELL_X_SI2E);
cell_y = ( (y_bound + next_y_cell) / CELL_Y_SIZE) ;
// проверить, находится ли в этом месте блок
if ((y_hit_type = world[(WORLD_ROWS-1) - cell_y] [cell_x] ) !=0)
{
// вычислить расстояние
dist_y = (xi - x) * inv_cos_table[view angle];
xi_save = xi;
yb_save = y_bound;
// закончить вычисление Y-пересечения
yray = INTERSECTION_FOUND;
casting--;
```

```
} // конец обработки попадания луча на блок
else
{
// вычислить следующее X-пересечение
xi += x_step,[view_angle];
} // конец оператора
else
} // конец проверки на отсечение по оси У
// перейти к следующей точке пересечения
x_bound += x__delta;
y_bound += y_delta;
}
//СЕКЦИЯ   6 ////////////////////////////////////
// выяснить, какая из стен ближе - вертикальная или горизонтальная
// и затем нарисовать ее
// Примечание: в дальнейшем мы заменим вертикальную линию на
// текстурный блок, пока же достаточно того, что есть
if (dist_x < dist_y)
{
sline(x,y,(long)xb_save,(long)yi_save, rcolor);
// вертикальная стена ближе горизонтальной
// вычислить масштаб и умножить на поправочный коэффициент
// для устранения сферических искажений
scale = cos_table[ray]*15000/(1e-10 + dist_x);
// вычислить координаты верха и низа
if ( (top    = 100 - scale/2) < 1) top =1;
if ( (bottom = top+scale) > 200) bottom=200;
// нарисовать фрагмент стены
if ( ((long)yi_save) % CELL_Y_SIZE <= 1 ) _setcolor(15);
else
_setcolor(10);
_moveto((int)(638-ray),(int)top);
_lineto((int)(638-ray),(int)bottom);
}
else // сначала надо нарисовать горизонтальную стену
{
sline(x,y,(long)xi_save,(long)yb_save,rcolor) ;
// вычислить масштаб
scale = cos_table[ray]*15000/(le-10 + dist_y);
// вычислить координаты верха и низа
if ( (top    = 100 - scale/2) < 1)
top = 1;
if ( (bottom = top+scale) > 200) bottom=200;
// нарисовать фрагмент стены
if (((long)xi_save) % CELL_X_SIZE <= 1 )
_setcolor(15) ;
else
_setcolor(2);
_moveto((int)(638-ray),(int)top);
_lineto((int)(638-ray),(int)bottom) ;
} //конец оператора else
//СЕКЦИЯ   7 ////////////////////////////////////
//отсечь следующий луч
if (++view_angle>=ANGLE_360)
{
// установить угол в 0
view_angle=0;
} // конец оператора if
} // конец цикла for по лучам
} // конец функции
```

Реализация алгоритма отсечения лучей разделена на семь основных частей, чтобы легче было понять предназначение каждой из них.

- Первая часть произведет инициализацию всех массивов и внутренних переменных. В этой части определяется направление взгляда игрока и на его основании вычисляется стартовый угол. Также в этой секции выбирается случайное число, задающее цвет трассируемого луча;
- Вторая и третья части вычисляют X- и Y-пересечения текущего луча с периметром ячейки, в которой находится игрок. После того как найдено первое пересечение как с осью X, так и с осью

Y, устанавливается значение нескольких переменных для определения траектории луча по отношению к осям координат. Информация, полученная в этих частях, используется в четвертой и пятой части;

- Четвертая и пятая части продолжают проверку пересечений. Каждое из пересечений с координатной осью проверяется на пересечение с объектом. Если это происходит, то вычисляется дистанция и запоминается для дальнейшего использования. Эта информация может быть использована для тексту рирования объектов. Хотя рассмотренный нами трассировщик лучей и не делает этого, он предоставляет достаточно информации для текстурирования. Например, если луч пересекает середину стены блока, это означает, что мы должны вывести на экран 32-ю вертикальную полосу соответствующей текстуры. Более подробно этот вопрос будет рассмотрен позднее, в главе, посвященной описанию игры Warlock;
- В шестой части заканчивается обработка луча. К этому моменту мы уже вычислили горизонтальные и вертикальные пересечения и запомнили расстояния до них. Следовательно, мы готовы нарисовать на экране соответствующую лучу вертикальную полосу. Для этого мы определяем, которое из пересечений находится ближе всего. Горизонтальная позиция для отрисовки соответствует номеру текущего луча и меняется в диапазоне от 0 до 319. Высота рисуемого фрагмента вычисляется на основании расстояния до игрока и с некоторыми корректировками для улучшения впечатления;
- Седьмая часть увеличивает текущий угол и осуществляет переход к первой части. Цикл выполняется до тех пор, пока все 320 лучей не будут отсечены.

Вообще, то, что мы написали - неплохая штука. Она не рисует фактур и не просчитывает освещение, но это несложно реализовать. Теперь, когда у нас появилась работающая программа, есть смысл поговорить об оптимизации.

Оптимизация отсекателя лучей

Отсекатель лучей, который мы написали, сложно назвать законченной программой. Она не делает множества вещей, но, несмотря на это, давайте поговорим как можно заетавить ее работать быстрее.

- Первое и обязательное условие — это использование целочисленной арифметики вместо операций с плавающей запятой. Это позволит увеличить скорость примерно в 2-4 раза;
- Во-вторых, мы можем многое оптимизировать в самом тексте программы на Си. Это поднимет быстродействие примерно на 20-50 процентов;
- Далее, мы можем разбить X- и Y-отсечения на две независимые программы. Это позволит производить вычисления параллельно;
- И, наконец, никакая нормальная программа не станет использовать графическую библиотеку Microsoft или разрешение 640x480.

Если использовать уже написанные функции для режима 13Ь, то выполнение программы ускорится в 10 раз и она начнет работать со скоростью 30 кадров в секунду. Но даже если скорость окажется около 15 кадров в секунду, то я буду рад.

Отрисовка дверей и прозрачных областей

Я надеюсь, вы видели, как открываются двери в Wolfenstein'e и DOOM'e. Когда они открыты, вы можете заглянуть в соседнюю комнату. С точки зрения трассировки лучей, при этом возникает множество проблем. С одной стороны, Дверь должна быть нарисована. С другой — должно быть нарисовано и то, что находится позади нее. Для решения этой проблемы луч трассируется до пересечения с дверью и затем проходит сквозь нее.

Чтобы понять, как это делается, давайте обсудим прозрачность с точки зрения возможности просмотра сквозь участки стен. Предположим, что вы отрисовываете текстуру стены, имеющей отверстия, позволяющие видеть то, что находится с другой стороны. Если нас там поджидает чудовище, мы должны его увидеть.

Это достигается следующим образом:

1. В процессе трассировки луч достигает сплошной стены.
2. Если стена содержит отверстия, пересечение все равно засчитывается и запоминается расстояние до игрока. Но трассировка луча продолжается дальше, до пересечения со следующей стеной.
3. Запоминается полученное пересечение.

Этот процесс может продолжаться до бесконечности, но сейчас давайте предположим, что он прекращается после пересечения второй стены. Как показано на рисунке 6.31, мы получаем два пересечения.

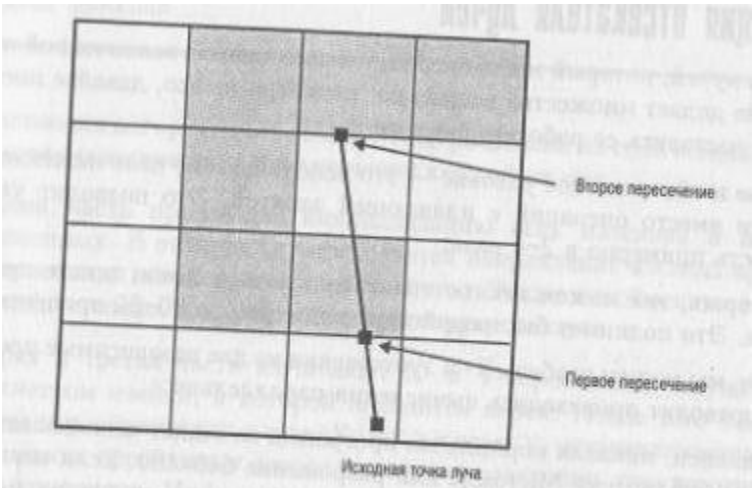


Рис. 6.31. *Трассировка луча после первого пересечения.*

Первое пересечение позволит отрисовать соответствующую вертикальную полосу. Однако второе пересечение тоже даст вертикальную полосу. Корректная отрисовка обоих вертикальных фрагментов осуществляется в соответствии с Алгоритмом Художника. Таким образом, вначале на основе полученных от второго пересечения данных будет отрисован более удаленный фрагмент стены, который затем будет частично перекрыт ближним фрагментом. Полученное таким образом изображение и будет выглядеть так, как если бы мы заглядывали в соседнюю комнату через окно.

Аналогично обрабатывается и открывание двери. Если дверь открывается вправо или влево, она частично уходит внутрь косяка. Таким образом, часть изображения физически исчезает с переднего плана. Для создания такого эффекта мы должны продолжить трассировку луча после пересечения с дверью. Однако в отличие от окон, убранная часть двери является полностью прозрачной, поэтому нам не надо запоминать оба пересечения. Мы вполне можем ограничиться отрисовкой изображения, полученного на основе пересечения луча с удаленной стеной. В общем, все эффекты типа окон, открывающихся дверей.и т. п. создаются тем же способом, что и рассмотренные примеры — сочетанием трассировки луча через прозрачные области и простого механизма Z-буферизации.

Освещение, тени и палитра

Я хочу вам рассказать, каким образом выполнены световые эффекты в DOOM и как это можно реализовать. Более подробно об этом можно будет прочитать в седьмой главе.

Дело в том, что в играх, подобных DOOM, используется куча разных уловок и трюков. Более того, редактор карт, с которым мы познакомимся позже, позволит работать с планами комнат, цветом и т. д. В общем, все, что связано с цветовыми эффектами в DOOM, сделано с помощью цветовой палитры.

В качестве примера я хочу вам показать один из возможных методов создания окружения с тенями, локальным и общим освещением. Давайте на минуту прервемся и поговорим о локальном и общем рассеянном освещении.

Рассеянное освещение

Рассеянное освещение создает находящийся в комнате источник света. Например, если у вас есть регулируемая лампа, то вы можете изменять уровень рассеянного освещения. Рисунок 6.32 показывает комнату при двух разных уровнях рассеянного освещения.

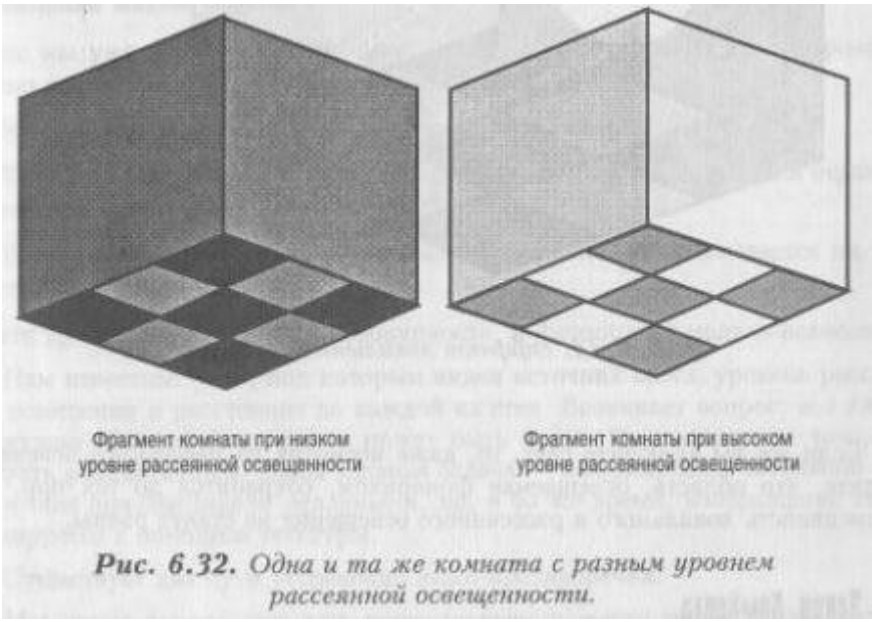


Рис. 6.32. *Одна и та же комната с разным уровнем рассеянной освещенности.*

Естественно, чем большим количеством-источников света вы располагаете, тем больше у вас возможностей воздействовать на рассеянную освещенность. Самый хороший пример общей освещенности - солнечный свет. Солнце находится от нас на столь большом, расстоянии, что падающие на землю лучи можно считать параллельными. Это создает некоторую среднюю или рассеянную освещенность, интенсивность которой зависит от высоты солнца над горизонтом.

Локальное освещение

Локальное освещение представляет собой концентрацию интенсивности света в определенном направлении с целью освещения ограниченной части помещения. Например, если, лежа в кровати, вы включите фонарик и направите его на стену, то увидите световое пятно (рис. 6.33). Если же вы включите свет, то, даже несмотря на рассеянное освещение, увидите, что область, освещаемая фонариком, сохранится до тех пор, пока интенсивность локального и рассеянного освещения не станут равны.

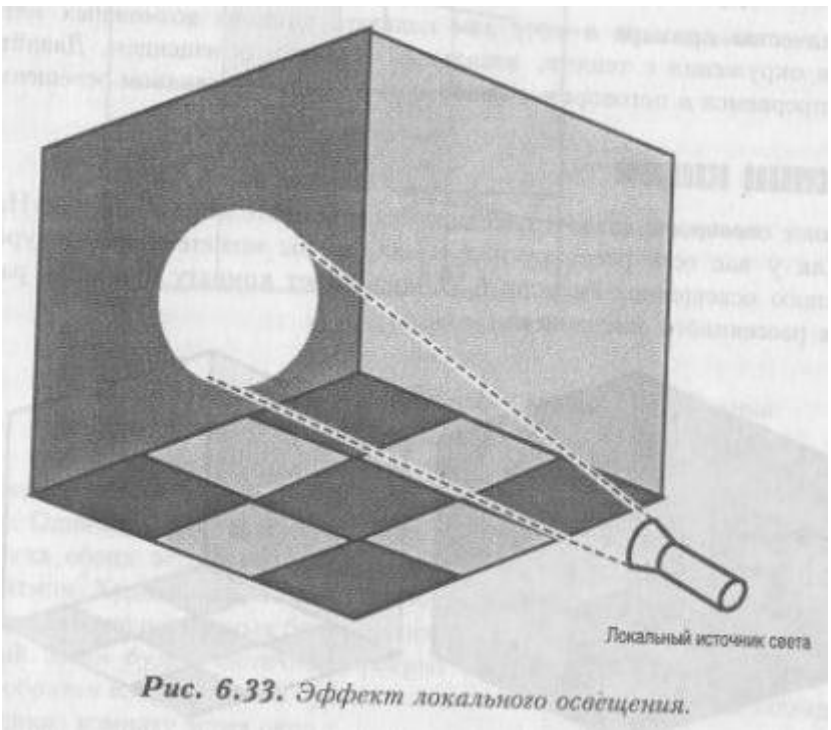


Рис. 6.33. *Эффект локального освещения.*

Закон Ламберта

Давайте поговорим об отражении и интенсивности. Если источник света направлен на поверхность под некоторым углом и наблюдатель смотрит на поверхность параллельно нормали, то интенсивность света изменяется в соответствии с законом Ламберта:

$I_d = I_i K_d \cos Q \quad 0 <= Q <= Pi/2$

где:

- I_d - результирующая интенсивность;
- I_i - интенсивность источника;
- K_d — коэффициент отражения материала;
- Q - угол между нормалью и направлением освещения.

Выражаясь нормальным языком, это означает, что отражение света возрастает, когда направление на источник света становится коллинеарным с нормалью к отражающей поверхности, или, иными словами, когда отражающая поверхность размещена прямо за источником света.

Закон обратных квадратов

Интенсивность света уменьшается обратно пропорционально квадрату расстояния до источника. То есть чем дальше вы находитесь от источника, тем меньше его интенсивность. Кстати, все это звучит очень правдоподобно.

Создание модели освещения

Сейчас мы уже изучили всю физику, которую надо знать для формирования хорошо выглядящей модели освещения. Мы знаем, что:

- Чем дальше находится источник, тем меньше света он дает;
- Если поверхность расположена под углом к источнику света, она отражает меньше света;
- Если изменяется уровень рассеянного освещения, это сказывается на всех объектах в комнате.

Все эти три фактора, взятые в совокупности, и формируют модель освещения.

Нам известны: угод, под которым виден источник света, уровень рассеянного освещения и расстояние до каждой из стен. Возникает вопрос: все ли это нам нужно? Может быть все, а может быть и нет. Это зависит от того, что называть реалистичным. Наша главная задача — сформировать затенение стен так, чтобы они выглядели реальными, но, в то же время, изображение стены формируется с помощью текстуры.

Существует два пути устранения этого противоречия.

Мы могли бы рассчитывать интенсивность каждого пикселя "на лету" и окрашивать его в соответствующий цвет. Единственная, возникающая при этом проблема, — недостаточное количество одновременно отображаемых цветов. Мы располагаем только 256 регистрами цвета, следовательно, одновременно на экране не может присутствовать больше 256 цветов. Это заставляет нас использовать цвет с большой осторожностью.

Тем не менее, это вполне работоспособный метод, дающий хорошие реэультаты. Я оставляю целиком на ваше усмотрение применимость к получающимся изображениям термина «реалистичные», но они достаточно хороши для видеоигр на ПК. Мы знаем, что должны изменять оттенок цвета стен в зависимости от угла, под которым они видны, и от их расстояния до игрока. Мы также знаем, что интенсивность окраски стен зависит от уровня рассеянного света. Теперь давайте сконцентрируемся на эффектах, связанных с углом обзора поверхности и расстоянием до нее.

Выполняя трассировку лучей, мы выводим на экран фрагмент текстуры. Ее вид определяется содержимым регистров RGB в таблице цветов (см. пятую главу). Для выполнения операции затенения мы должны знать угол, образуемый поверхностью с направлением взгляда, и расстояние до нее. К счастью, нам известно и то и другое. Теперь осталось только выяснить, как использовать имеющуюся информацию для формирования затенения.

На практике нам нет необходимости использовать оба параметра в алгоритме затенения. Мы можем использовать или расстояние от стены до игрока или угол между направлением взгляда и поверхностью стены для получения вполне реалистичного изображения. Единственная проблема состоит в том, как получить все возможные эффекты затенения с помощью только 256 цветов?

Для решения этой проблемы мы можем создать палитру со следующими свойствами:

- Первые 16 цветов - стандартные цвета EGA;
- Дополнительные цвета помещаются в следующие 56 ячеек палитры. Это те цвета, с которыми мы будем работать, и единственные цвета, присутствующие в игре. Эти 56 цветов должны быть выбраны так, чтобы их хватило в качестве базовых для изображения всех игровых объектов. Более того, эти цвета будут самыми яркими в игре и они должны создаваться с учетом этого факта;
- Теперь некоторая хитрость. Оставшиеся 184 цвета разбиваются на 3 банка по 56 цветов в каждом и банк из 16 цветов в конце палитры.

Три дополнительных банка по 56 цветов будут заполняться в процессе работы программы и использоваться механизмом формирования затенения. Дополнительный банк из 16 цветов используется для анимации палитры. Таким образом, палитра будет выглядеть как Представлено в таблице.

Таблица 6.1. Цветовая палитра для формирования затенения.

Регистры цвета	Функция

0 – 15	Базовые цвета EGA
16 - 71	Первичные цвета
72 – 127	Вторичные затененные производные первичных цветов
128-239	Третичные затененные производные
240 – 255	Дополнительные цвета для цветовой ротации и т.п.

Механизм затенения работает следующим образом: при отрисовке каждого вертикального фрагмента к значениям его пикселей добавляется некая константа. Например, если отрисовывается текстура из одного цвета с номером 16, то производные цвета будут иметь значения 16+56, 16+2x56 и 16+3x56 (или 72, 128 и 184 соответственно). В общем, механизм затенения берет за основу цвета текстуры и модифицирует их за счет использования других регистров цвета, номера которых определяются как сумма исходного цвета и константы. Если объект нарисован в первых 16 цветах, добавление константы не производится. Теперь мы имеем по четыре варианта каждого из 56 первичных цветов. Этого вполне достаточно для формирования реалистичного изображения. Три «затененных», банка цветов заполняются во время работы программы путем некоторого уменьшения интенсивности базовых цветов. Таким образом, всего получаются четыре банка из одних и тех же цветов с убывающей интенсивностью.

Теперь возникает вопрос, как пользоваться полученной таблицей и как готовить битовые карты? Когда вы будете рисовать свои битовые карты, рисуйте те объекты, которые впоследствии планируете затенять, с использованием 56 первичных цветов. Те объекты, которые затеняться не будут, рисуются в первых 16 цветах — они не затрагиваются механизмом затенения.

Кстати, так же создается и эффект локального освещения. Освещенные или светящиеся области рисуются с использованием первых 16 цветов и, соответственно, их цвет не изменяется при затенении.

Последняя проблема состоит в том, насколько мы должны затенять цвет. Как я уже говорил раньше, мы можем использовать или угол между поверхностью и лучом зрения или расстояние до игрока. Я рекомендую попробовать оба варианта и выбрать тот, который вам больше понравится. Что бы вы ни использовали, эта величина должна быть разбита на зоны. Если величина попадает в первую зону, используйте первую производную цвета, во второй зоне — вторую и т. д. Например, вы можете решить, что если угол составляет меньше 20°, то используется первая производная, если он оказывается в диапазоне 20-300 — вторая и т. д.

Та же логика может быть применена и к расстоянию. Надо только учитывать, что интенсивность цвета падает нелинейно. В качестве примера можно привести разбивку расстояния на зоны, представленную в таблице 6.2.

Таблица 6.2.

Дистанция (условных ед.)	**Банк цвета**
0-20	Банк 0(цвета 16 – 71)
20-50	Банк 1(цвета 72 – 127)
50-100	Банк 2(цвета 128 – 183)
100-бесконечность	Банк 3(цвета 184 – 239)

Аналогичная таблица может быть создана и для углов. К настоящему моменту мы еще не

готовы писать программу, реализующую механизм затенения, так как для этого требуется очень мощная оптимизация. Однако я надеюсь, что вы поняли идею.

Коротко повторим: цветовая палитра разбивается на шесть областей.

- Первая и последняя области содержат цвета, не используемые при затенении;
- Четыре средних области представляют собой банки по 56 цветов и каждый из этих банков представляет собой производные одних и тех же первичных цветов.

Когда изображение выводится на экран, каждый пиксель рисуется наиболее ярким цветом (банк 0), к которому добавляется некоторая константа, для формирования требуемой производной. Эта константа зависит от модели освещения, определяемой либо расстоянием от объекта до игрока, либо углом, под которым игрок смотрит на поверхность.

Вот и все.

ИТОГ

В этой главе мы изучили основы трехмерной графики, поговорили о математических основах трехмерных трансформаций и методах, используемых для обсчета трехмерных образов (удаление невидимых поверхностей и т. д.).

Кроме этого, мы узнали о двух наиболее важных с точки зрения создания красивых DOOM-образных игр методах: о трассировке и отсечении лучей. Мы даже написали реализацию алгоритма отсечения лучей.

Позже, когда настанет пора написать игру Warlock, мы еще раз вспомним эти методы и реализуем все, о чем мы говорили в этой главе, в полном объеме. А пока давайте перейдем к следующей главе. Кстати, пока не забыл, если вы что-то не поняли, прочитайте и осмыслите главу еще раз.

УЛУЧШЕННАЯ БИТОВАЯ ГРАФИКА И СПЕЦИАЛЬНЫЕ ЭФФЕКТЫ

Границу между просто хорошей и отличной игрой провести довольно сложно. Ведь разнообразие алгоритмов, сюжетов и звуков, применяемых в играх, невелико. В чем же причина того, что одна игра становится бестселлером, а другая - нет? Все дело в нюансах: более искусный переход от одной картинки к другой, мягкая прокрутка, точная синхронизация и т. д. Чтобы добиться этого, мы должны постоянно оттачивать наше мастерство, как воин - свой меч.

В этой главе вам встретятся некоторые Действительно интересные (и, вообще-то, довольно простые) программы. Они создают такие же сложные и реалистичные эффекты, как те, которые вы наверняка видели в компьютерных играх. Также на этих страницах вы узнаете о таком сложном приеме работы с растровой графикой, как масштабирование. Итак, в этой главе:

- Ускорение процесса двоичного кодового преобразования (бит-блиттинга);
- Применение логических операций;
- Кодирование прозрачности;
- Битовое отсечение;
- Контроль столкновения спрайтов;
- Дублирующая буферизация;
- Использование сигнала вертикальной синхронизации;
- Мультипликация с помощью цветовых регистров;
- Освещение ваших игр;
- Связь мультипликации с контекстом;
- Мультипликационное движение («animotion»);
- Прокрутка;
- Специальные эффекты;
- Текстуры;
- Масштабирование растровых изображений;
- Повороты растровых изображений.

Итак, больше дела, меньше слов!

Ускорение процесса двоичного кодового преооразовзния (бит-блиттинга)

Если подумать о типах разрабатываемых нами компьютерных игр, то станет очевидно, что основной упор нам придется делать на графику.

При работе с такими играми (трехмерные игры, подобные Wolfenstein 3D или DOOM) компьютер большую часть времени тратит на прорисовку стен в трехмерном пространстве или на изображение двухмерных образов спрайтов, представляющих игровые объекты. Мы обсудили изображение в трехмерном пространстве в шестой главе, «Третье измерение». Поэтому теперь я хотел бы остановиться на вопросе оптимизации вывода спрайтов, поскольку это нам особенно понадобится при создании игр данного типа.

В предыдущих главах мы разобрались с понятием бит-блиттинга (глава пятая, «Секреты VGA-карт») и рассмотрели, как передается растровое изображение из одной области памяти в другую. Держу пари, что 90 процентов времени при создании игр уходит на то, чтобы придумать, как ускорить эти процессы. В восемнадцатой главе, «Техника оптимизации», мы обсудим общую теорию оптимизации и разберем несколько примеров использования соответствующей техники после завершения работы над игрой. (Не забывайте о том, что все оптимизационные трюки следует применять на заключительной стадии разработки игры!) Вы должны четко представлять, что для каждой новой игры необходимо заново создавать и реализовывать подходящие алгоритмы битового замещения; более того, вы можете исгюльзовать от двух до пяти различных способов бит-блиттинга в одной и той же игре. При этом каждый из них будет предназначаться для вполне конкретного случая. Например, вы можете обнаружить, что можно оптимизировать функцию бит-блиттинга для объектов, которые на протяжении всей игры остаются неподвижными. Если это так, создайте два преобразования: одно — для движущихся объектов, другое — для стационарных. Код для каждого процесса бит-блиттинга занимает примерно один-два килобайта ияи даже меньше.

Рассмотрим еще один пример возможной оптимизации. Естественно, что чтение данных вашей функцией бит-блиттинга определенным образом зависит от способа их хранения. И возможно, преобразование будет происходить быстрее, если представить данные в другом виде. Например, вы можете потерять эффективность, сохраняя значения пикселей построчно (в виде двухмерной матрицы), а не по столбцам (по 16 бит). Если иная форма хранения данных дает выигрыш в производительности для некоторых из ваших алгоритмов бит-блиттипга, напишите для них процедуру предварительного преобразования этих данных.

В любом случае, думайте о том, как максимально ускорить процесс бит-блиттинга. Ведь в компьютерной игре на экране присутствуют в среднем от 2 до 20 объектов размером примерно 32x32 пикселя, а это достаточно много для работы в реальном времени.

Примечание

Как вы знаете, я привожу в этой книге некоторые алгоритмы бит-блиттинга. Пожалуйста, не

используйте их в ваших компьютерных играх! Они написаны исключительно в демонстрационных целях. Вы должны для каждого конкретного случая найти свой вариант.

Рассмотрим еще один важный аспект бит-блиттинга объектов, который вы сможете с выгодой использовать. Если вы твердо знаете, что размер ваших объектов будет фиксированным (например, 16x16 или 32x32), то можете написать отдельный алгоритм бит-блиттинга, специально оптимизированный для каждого из этих размеров. Помните: не существует общих правил быстрого написания компьютерных игр. Нет уже готовых алгоритмов и написанных кем-то библиотек. Вы должны будете использовать по максимуму весь свой творческий потенциал. Только, не пугайтесь. Даже самая замысловатая игра использует алгебру и геометрию не более чем на уровне высшей школы. Это, конечно, не значит, что вы не можете использовать сверхсложные алгоритмы и забираться в дебри, высшей математики. Однако в результате может оказаться, что двенадцатилетний подросток найдет путь в 100 раз более быстрый и эффективный только потому, что он не такой умный как вы!

Применение логических операций

Для того чтобы переместить образ (процесс блиттинга) на экран или даже вынести его за пределы рабочей области экрана, можно выполнять логические операции с исходными и результирующими пикселями. Если вы внимательно посмотрите на функцию DrawSprite () в Листинге 5.13 пятой главы, «Секреты VGA-карт», то увидите, что она на самом деле не просто замещает данные, а предварительно проверяет их тип:

- Если пиксели «непрозрачные (то есть они не черного цвета), то на экране происходит замена исходных данных соответствующими пикселями;
- Если пиксели «прозрачны», то экранные данные остаются без изменений.

Все происходит так, как если бы в объекте имелись дырки, через которые можно смотреть. Для получения "прозрачности" нужно поместить внутрь цикла бит-блиттинга условный оператор примерно следующего вида:

```
if ((data=work_sprite[work_offset+x]))
            video_screen[offset+x]=data;
```

Оператор IF выполняется много раз. Если быть точными, то число повторов этой операции равно произведению высоты спрайта на его ширину. Так для образа 64x64 это составит 4096 раз. Не слишком ли много?!

Однако в некоторых случаях можно отказаться от использования оператора IF и выводить спрайт без проверки пикселей на «прозрачность». Если нарисованный нами спрайт имеет точно прямоугольную форму и целиком занимает объем 64x64 пикселя, исходные данные без ущерба можно замещать на реэультирующие (конечно, если внутри самого изображения нет «прозрачных» областей). Увы, так бывает довольно редко! Конечно, в подобных случаях мы можем применять и такой вариант блиттинга (например, при текстурировании стен), но, тем не менее, попробуем найти «золотую середину» между использованием оператора IF и функции memcpy ().

Один из способов размещения данных на экране состоит в употреблении поразрядных логических операций типа OR, XOR, AND и NOT. Таким образом, мы можем воспользоваться ими вместо оператора IF, поскольку они быстрее обрабатываются центральным процессором. Осталось только выяснить совсем немного — какой же из логических операторов применить и что он будет делать? Вспомним, как представлены данные в режиме 13h. Каждый пиксель задается одним байтом, значение которого используется в качестве индекса в таблице выбора цвета. Если мы начнем производить логические операции с исходными и результирующими пикселями, то изменим индекс цвета, а, следовательно, и сам цвет, чего нам совсем не нужно!

Рассмотрим несколько примеров. Допустим, мы отображаем пиксель красного цвета (со значением 10) на экран в точку (100,100), в которой уже находится розовый пиксель (56). В результате отображения мы хотели бы увидеть на экране наш красный пиксель (10) в положении (100,100). Однако при использовании имеющихся в нашем распоряжении логических операций, вместо этого мы получим значения, приведенные в таблице 7.1.

Таблица 7.1. Результаты логических операций.

Дано: 56 - 00111000b и 10 = 00001010b:

Источник(битовый образ)	**Операция**	**Результирующие данные (данные экрана)**	**Результат**
00111000	OR	00001010	00111010 =58
00111000	AND	00001010	00001000 =8

00111000	XOR	00001010	00110010 =50

Нетрудно заметить, что в результате любой из логических операций изменяется индекс цвета. Иногда это может совпадать с нашим желанием, но в данном случае это не то, что мы хотели получить, замещая «непрозрачные» (ненулевые) пиксели экрана новым значением.

Однако нам никто не запрещает совершить более одной логической операции с исходными и результирующими данными для достижения желаемого результата. Мы можем делать все, что захотим, но главное при этом — не потерять тот выигрыш в быстродействии, который мы получаем при использовании логических операций. Ведь оператор IF осуществляет только два действия: сравнение и переход в нужное место. Так не окажется ли применение оператора IF более целесообразным, нежели многократное повторение логических операций? Чтобы выяснить это, давайте сравним коды, реализующие оба способа (Листинги 7.1 и 7.2), и посмотрим, как обстоят дела на самом деле.

Листинг 7.1. Дизассемблирование оператора IF.

```
; File if.c   ; # include <stdio.h>
;
; char far *source, *dest;   //области исходных
//и результирующих данных
; main()
;{
; Line 8
_main:
*** 000000  c8 00 00 00       enter OFFSET L00181,OFFSET 0
*** 000004  56                push si
*** 000005  57                push di ; index = fffc ; data = fffa
;
;int index;
; Line 10
; unsigned data;
; Line 11
; // оператор if
; if (data=source[index]) ; Line 15
*** 000006  8b 46 fc          mov  ax, WORD PTR -4[bp]
*** 000009  8b 1e 00 00       mov  bx,WORD PTR  source
*** OOOOOd  8b 0e 02 00       mov  ex,WORD PTR  source+2
*** 000011  03 d8             add  bx,ax
*** 000013  8e cl             mov  es,cx
*** 000015  26 8a 07          mov  al,BYTE PTR es:[bx]
*** 000018  98                cbw
*** 000019  89 46 fa          mov  WORD PTR -6[bp] ,ax
*** 0000lC  3d 00 00          cmp  ax,OFFSET 0
*** 0000lf  75 03 e9 00 00    je   L00180
;   dest[index]=data;
; Line 16
*** 000024  8b 46 fa          mov  ax,WORD PTR -6[bp]
*** 000027  8b 4e fc          mov  ex,WORD PTR -4[bp]
*** 00002a  8b Ie 00 00       mov  bx,WORD PTR _dest
*** 00002e  03 d9             add  bx,cx
*** 000030  88 07             mov  BYTE PTR [bx],al ;
} // end main
; Line 18
L00180;
; Line 18
L00177:
*** 000032  5f                pop  di
*** 000033  5e                pop  si
*** 000034  c9                leave
*** 000035  cb
ret OFFSET 0 Local Size: 6 ; Line 0
```

Листинг 7.2. Дизассембдирование логической операции OR.

```
; File or.c
; #include <stdio.h> ;
; char far *source, *dest; // области исходных
```

```
// и результирующих данных
;
; main()
; {
; Line 8
     _main:
*** 000000 c8 00 00 00        enter OFFSET L00180,OFFSET 0
*** 000004 56                 push si
*** 000005 57                 push di ; index = fffc ;data = fffa
;int index;
;Line 10 ;unsigned data;
;Line 11
// логический оператор (OR)
dest[index]=data | source[index];
; Line 15
*** 000006 8b 46 fc          mov ax,WORD PTR -4 [bp]
*** 000009 8b Ie 00 00       mov bx,WORD PTR _source
*** 00000d 8b 0e 02 00       mov ex,WORD PTR _source+2
*** 000011 03 d8             add bx,ax
*** 000013 8e c1             mov es,cx
*** 000015 26 8a 07          mov al,BYTE PTR es:[bx]
*** 000018 98                cbw
*** 000019 0b 46 fa          or ax,WORD PTR -6[bp]
*** OOOOlc 8b 4e fc          mov ax,WORD PTR -4[bp}
*** OOOOlf 8b Ie 00 00       mov bx,WORD PTR _dest
*** 000023 03 d9             add bx,cx
*** 000025 88 07             mov BYTE PTR [bx] , al
*** 000027 8b 4e fc          mov ex,WORD PTR -4[bp]
*** 00002a 8b Ie 00 00       mov bx,WORD PTR _dest
*** 00002e 03 d9             add bx,cx
*** 000030 88 07             mov BYTE PTR [bx],al
;
; } // end main
; Line 17
; Line 17
L00177:
*** 000032 5f                pop di
*** 000033 5e                pop si
*** 000034 c9                leave
*** 000035 cb                ret OFFSET 0 Local Size: 6
; Line 0
```

Если вы внимательно посмотрите на Листинги 7.1 и 7.2, то увидите, что в каждой программе я выделил некую область. Именно эти куски и определяют принципиальное различие приведенных программ. Мы видим, что код, в котором используется логическая операция OR, почти в два раза короче кода с оператором IF. Однако, мы должны прикинуть и время выполнения обоих фрагментов. Это довольно сложно, так как мы не знаем, какой процессор будет использован: 386, 486 или 586. Тем не менее, мы можем приблизительно подсчитать, что код, содержащий операцию OR, будет выполняться почти в два раза быстрее, чем код, в котором используется IF. Однако если нагородить несколько операторов OR, можно легко потерять эффективность этого приема. Поэтому имеет смысл заменять оператор IF только одной, максимум двумя логическими операциями. Следовательно, мы должны найти возможность получения требуемого результата с применением единственной логической операции.

Один из путей такой. Мы можем воспользоваться разнообразными специальными эффектами и нетривиальными манипуляциями, изменяющими с помощью логических операций значение пикселя заданным образом. Например, можно создать такую таблицу выбора цветов, чтобы в результате логических операций получившийся новый индекс цвета имел то же значение RGB, что и исходные данные, а его числовое значение было бы логической комбинацией исходных и результирующих данных.

Попробуем создать цветовую модель для того, чтобы проиллюстрировать это. Пусть в нашей игре фон имеет 16 цветов с индексами 0, 16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 176, 192, 200 и 224. Изображение объекта (или спрайт) также содержит 16 цветов, только индексироваться они будут от 0 до 15. Следовательно, когда к исходному и результирующему пикселю применяется логическая операция OR, четыре младших бита результата всегда будут определять цвет фона, в то время как старшая тетрада задаст цвет объекта.

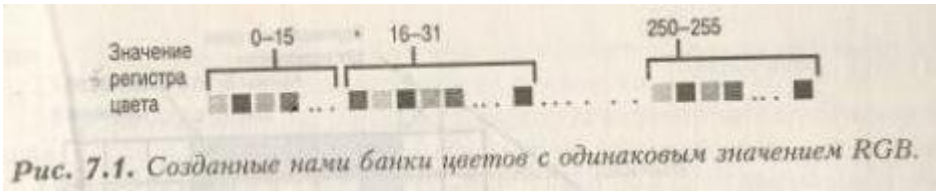


Рис. 7.1. Созданные нами банки цветов с одинаковым значением RGB.

Предположим, мы последовательно скопируем первые 16 значений в таблицу выбора цветов и создадим таким образом несколько банков, содержащих по 16 тонов с одинаковыми значениями RGB, как это показано на рисунке 7.1.

В таком случае банк 0 будет иметь диапазон от 0 до 15, банк 1 — от 16 до 31 и т. д. Каждый банк будет содержать одинаковые значения RGB, поэтому, когда данные фона примут нулевое значение, цвет объекта не изменится. Точнее:

```
0 OR (0-16) = (0-16)
```

Однако интереснее, когда исходные данные принимают ненулевое значение. Предположим, что они имеют значение 5 (или цвет 6, с точки зрения исходного изображения), в то время как данные фона равны 32 (или цвет 2). Применив К этим значениям логическую операцию OR, мы получим число 37. Это соответствует пятому цвету в третьем банке. Цвет 37 имеет такое же значение RGB, как цвет 6 или индекс 5, так как в каждый банк мы копировали по 16 цветов. (Не пытайтесь уменьшать количество цветов в игре, их и так всего 16! Но в то же время важно постараться с умом использовать таблицу выбора цвета, чтобы по возможности ускорить процесс бит-блиттинга).

Скорее всего, мы не захотим использовать эту модель в чистом виде. Однако она дала нам в руки новый инструмент и идею, которую мы сможем использовать где-нибудь еще.

Кодирование прозрачности

Как я уже говорил в предыдущей главе, скорость бит-блиттинга можно увеличить, если использовать другие структуры данных. Наша основная проблема в отображении прямоугольных спрайтов заключается в реализации «прозрачности»: когда «прозрачный» пиксель (обычно 0, или черный) встречается в исходном изображении, пиксель фонового изображения остается неизменным (то есть не замещается). Эта операция выполняется с помощью оператора IF, и поэтому на нее тратится достаточно много времени.

Еще одно возможное решение данной проблемы заключается в применении к данным спрайта группового кодирования (run-lenght encoded, RLE), после которого последовательности «прозрачных» пикселей оказываются сгруппированными, как показано на рисунке 7.2.

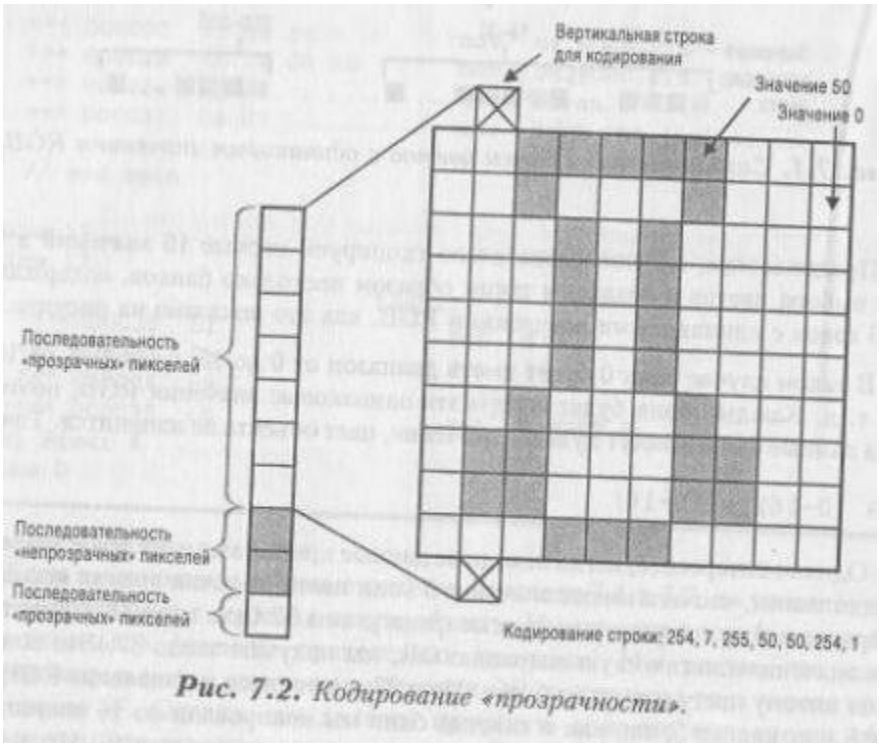


Рис. 7.2. Кодирование «прозрачности».

Если мы разобьем наш отображаемый объект на столбцы или строки, то сможем проанализировать последовательности «прозрачных» пикселей и пометить их соответствующим образом. В этом случае мы могли бы включать и выключать «прозрачность» подобно тому, как с помощью переключателя зажигаем и гасим лампочку. Это происходит в несколько десятков раз быстрее, чем если бы мы тысячи раз повторяли оператор IF. Как осуществить групповое кодирование — решать вам, однако, в качестве отправной точки я предлагаю следующее:

- Убедитесь, что данные спрайта записаны в формате:

столбец 1, столбец 2, ..., столбец N

где каждый из столбцов состоит из «прозрачных» и «непрозрачных» пикселей;

- Чтобы обозначить каждый из типов пикселей, задайте флаг, например, 255 и 254;
- Следующий за флагом элемент данных должен определять длину последовательиости в байтах;
- Затем перечислите значения пикселей выделенной последовательности.

Декодировать столбец данных можно будет по такому алгоритму:

```
ЕСЛИ
255: непрозрачный
254: прозрачный
тогда
   (255 | 254) , количество байт данных, данные
```

Этот шаблон будет повторяться много раз. Например, первый столбец на рисунке 7.2 будет кодироваться так:

```
254, 7, 0, 0, 0, 0, 0, 0, 0, 255, 2, 50, 50, 254, 1, 0
```

Алгоритм, который мы должны написать, будет для каждого столбца пропускать «прозрачные» пиксели и рисовать только «непрозрачные».

Затем можно провести дополнительную оптимизацию. Мы знаем, что «прозрачным» пикселям соответствуют нулевые значения данных (или любой другой индекс цвета, представляющий «прозрачность»), так зачем тратить место на избыточные нули 8 группах данных спрайта? Мы их можем отбросить. Новый формат будет содержать «непрозрачные» данные, флаги и значения длин последовательностей. В этой схеме возникает лишь одна небольшая проблема: мы не сможем использовать числа 254 и 255 как индекс цвета, поскольку эти значения уже использованы в качестве флагов контроля. (Однако я не думаю, что отсутствие двух цветов из 256 смертельно, не так ли?)

Опять-таки, это только идея, как ускорить процесс блиттинга. Вы можете использовать какой-то другой метод кодирования. Например, можно разбивать изображение не на столбцы, а на строки, но в этом случае могут возникнуть сложности при попытке наложения мира бликов на мир объекта. (О причинах этого вы можете прочесть в шестой главе, «Третье измерение».) Как бы там ни было, ваша копилка пополнилась еще одним секретом. Конечно же, вам понадобится написать дополнительную функцию преобразования стандартного битового изображения таким образом, чтобы ваша функция Draw_Sprite () смогла его обработать.

В этом разделе мы обсудили несколько идей улучшения бит-блиттинга и перемещения спрайтов. Я еще раз хотел бы подчеркнуть, что все сказанное — это только идеи. Я просто попытался продемонстрировать вам новый взгляд на данную проблему. Теперь вы знаете несколько

путей ускорения процесса бит-блиттинга, но, по мере того как вы будете набираться опыта и знаний, у вас будут появляться все новые и новые пути. (Я знаю в 20 раз более быстрый способ осуществления бит-блиттинга, чем тот который мы здесь обсуждали, однако он применяется только в специальных случаях и основан на таком алгоритме оптимизации, что я потратил кучу времени, чтобы разобраться с ним — но все же я его победил!) Таким образом, существует далеко не один вариант ускорения блиттинга. И для того чтобы достигнуть уровня игр типа Wolfenstein или DOOM, вы должны быть очень искусны и изобретательны в программировании

Битовое отсечение

Битовое отсечение означает вырезание растрового изображения краями экрана или другими границами. Мы не будет обсуждать здесь общие проблемы отсечения. Нас интересует только прямоугольный или оконный тип отсечения. Даже в играх наподобие Wolfenstein или DOOM используется только прямоугольное отсечение границами экрана или прямоугольниками, находящимися внутри экранной области. На рисунке 7.3 показан пример отсечения растрового изображения.

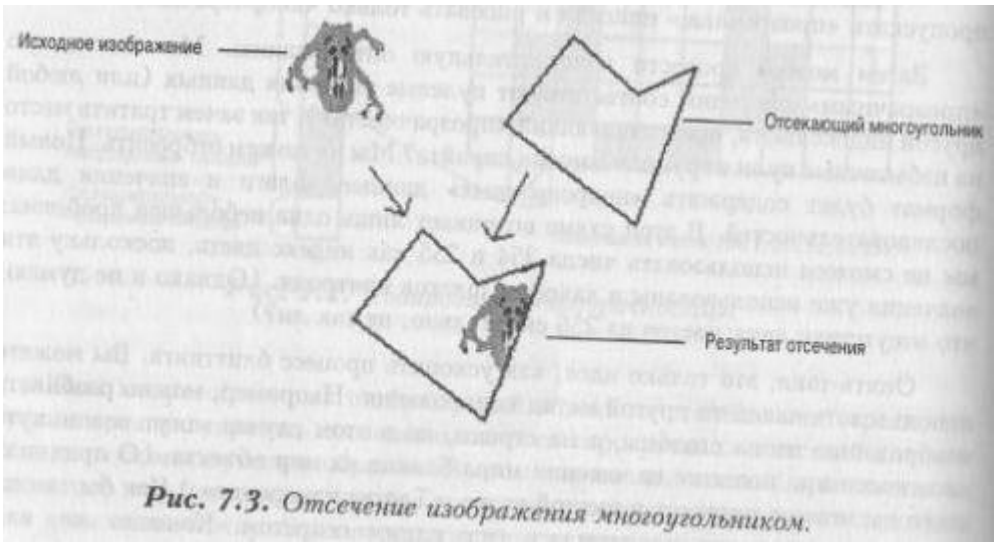


Рис. 7.3. *Отсечение изображения многоугольником.*

Каким же образом осуществляется отсечение прямоугольного изображения? И, прежде всего, нужно ли оно нам вообще? На этот вопрос можно ответить и «да» и «нет». Если во время игры изображения ваших объектов и персонажей никогда не выходят за границы экрана, то применять отсечение не обязательно. В этом случае необходимо только выполнять логический контроль за тем, чтобы при движении или трансформации объектов образ никогда не выходил за пределы экрана.

Однако, это не всегда допустимо. Например, в трехмерном DOOM'е монстры часто видны на игровом экране только частично. Скажем, видна только правая половина его тела. Это значит, что его левая часть должна быть отброшена при выводе образа и следует применить какой-либо алгоритм битового отсечения.

Мы всегда должны отсекать растровое изображение в тех случаях, когда оно заезжает за пределы экрана (или за пределы других установленных нами границ). Например, в трехмерной игре Wolfenstein (равно как и в DOOM) играющий может менять размер окна с изображением, но картинка никогда не будет выходить за пределы этого окна, так как она отсекается его краями.

Как же мы можем произвести отсечение изображения? Существует два пути:

- Мы можем проверять каждый отдельный пиксель, находится ли он внутри отображаемой области. Такое отсечение называется *отсечением пространства образа.* Другими словами, мы обрабатываем каждый из пикселей внутри всего образа и принимаем решение, рисовать его или нет. Эта техника не принимает во внимание геометрические свойства объекта. Она очень медлительна. (И я подозреваю, что именно такой метод применила фирма Microsoft в своей графике, потому что я никогда не видел блиттинг, работающий более медленно, чем в Microsoft's Graphics Library!) Посему данным методом мы пользоваться не будем;
- Мы можем учитывать такие геометрические свойства объекта, как его размер, его форма и его положение по отношению к области отсечения. Этот способ называется *отсечением области объекта.*

Мы воспользуемся вторым приемом. Он довольно прост, по крайней мере, в том виде, в котором я собираюсь показать его на примере. Мы, как всегда, хотим до&тичь максимума производительности и эффективности. А так как отсечение замедляет блиттинг, нужно постараться свести потери времени к минимуму.

Во-первых, мы должны провести предварительный тест и выяснить: *находится ли объект внутри ограниченной области?* Например, пусть наш объект 16x16 и он находится в точке (1000,1000). Экран ограничен. Поскольку экран имеет размеры 320x200, совершенно ясно, что этот образ окажется невидим, и никакое отсечение здесь не поможет. Таким образом, мы должны проверять каждый объект на предмет того, будет ли он виден хотя бы частично. Алгоритм 7.1 выполняет данный тест.

Алгоритм 7.1. Тест видимости объекта.

```
// Пусть объект размером Width x Height находится в точке (x,y).
// Размер экрана - Screen Width x Screen Height.
// Для каждого объекта производим следующую операцию:
if (X+Width>0 and X<Screen_Width and Y+Height>0 and Y<Screen_Height)
```

```
then
полностью или частично видим
goto отсечение
else
нет, не видим
goto следующий объект
```

Алгоритм 7.1 не особенно эффективен. Однако он выполняет свою функцию: если объект полностью или хотя бы частично должен появиться на экране, то нужно выполнить отсечение, после чего можно вывести изображение на экран. В противном случае проверяется следующий объект.

После того как мы приняли решение использовать отсечение, мы просто подсчитываем новые размеры и положение границ вырезанного изображения. Если вы попробуете это себе представить, то легко увидите, что в большинстве случаев отсечение может проводиться не по всем четырем, а только по двум смежным сторонам рисунка, например, по верхней и правой. А в некоторых случаях достаточно отсекать образ вообще только по одной стороне. Это выполняется для всех растровых изображений, имеющих размер меньше целого экрана.

Если же растровое изображение больше экрана хотя бы по одному измерению, то идея «вылетает в трубу». Так обычно бывает, когда мы масштабируем спрайт, имитируя приближение игрока к объекту. Но пока мы будем предполагать, что спрайты всегда меньше игрового поля экрана.

Если вы изучали функцию Draw_sprite() пятой главы или другие ее модификации, то, должно быть, заметили, что в ней выполняется вложенный цикл FOR, в котором битовый образ объекта рисуется по столбцам. Каждый столбец начинается с X-координаты спрайта и спускается вниз по координате Y. Таким образом, мы должны изменить механизм этого цикла так, чтобы принять во внимание новые отправные и конечные координаты (x,y) и изменить исходное местоположение спрайта, если одна из координат выходит за пределы экрана. Не забывайте о том, что если координата X или Y (либо обе) выходят за пределы экрана, это еще не значит, что ваш объект вообще невидим. На рисунке 7.4 изображен иллюстрирующий это пример.

Давайте изменим функцию Draw_sprite(), чтобы отсечь данные, не попадающие на экран. Новая версия будет работать на 5 процентов медленнее, однако она всегда будет корректно рисовать спрайты. Листинг 7-3 содержит ее исходный текст.

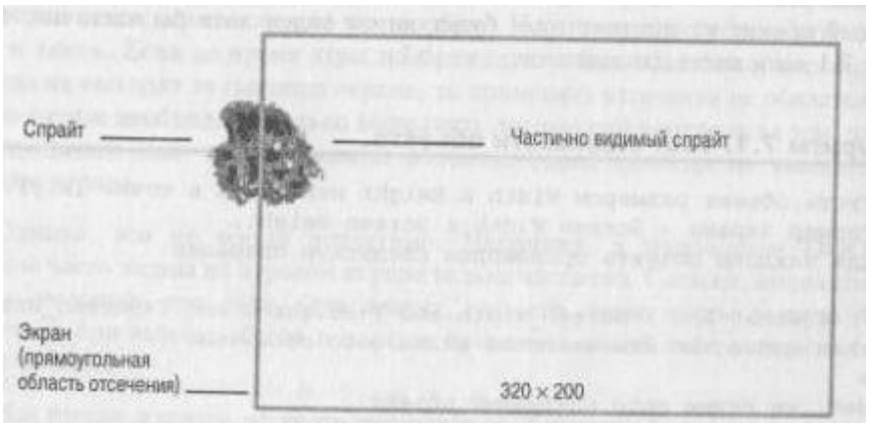


Рис. 7.4. Частично видимый объект.

Листинг 7.3. Новая функция Draw_Sprite() с отсечением.

```
void Draw_Sprite_Clip(sprite_ptr sprite)
{
// Эта функция рисует спрайт на экране и отсекает области,
// выходящий за границы экрана. Функция написана в
// демонстрационных целях.
char far *work_sprite;
int work_offset=0,offset,x_off,x,y,xs,ys,xe,ye, clip_width, clip_heigth;
unsigned char data;
// Получить указатель на спрайт
xs=sprite->x;
ys=sprite->y;
// Расчет координат прямоугольника, описывающего спрайт
xe=xs+sprite_width-1;
ye=ys+sprite_height-1;
// Проверка полной невидимости спрайта
// (то есть лежит ли он полностью за пределами экрана)
if((xs>=SCREEN_WIDTH) || (ys>==SCREEN HEIGHT) | |
(xs<=(-sprite_width))||(ys<=(-sprite_height)))
{
return;
} // конец оператора if

// Спрайт частично видим, следовательно, необходимо
// рассчитать рисуемую область
// Отсечение по координате X
```

```
if(xs<0)
xs=0;
else
if(xe>=SCREEN_WIDTH) xe=SCREEN_WIDTH-l;
//Отсечение по координате Y
if(ys<0)
ys=0;
else
if(ye>=SCREEN_HEIGHT) ye=SCREEN_HEIGHT-1;
// Расчет новой высоты и ширины
clip_width=xe-xs+l;
clip height=ye-ys+l;
// Расчет рабочего смещения на основе нового начального
// значения координаты Y
work_offset=(sprite->y-ys)*sprite_width;
x_off=(xs-sprite->x);
// Построение усеченного спрайта
// Для упрощения дадим указателю на спрайт новое имя
work_sprite = sprite->frames[sprite->curr_frame];
// Расчет смещения,спрайта в видеобуфере
offset = (ys<< 8) + (ys<< 6) + sprite->xs;
for (y=0; y<clip_height; y++)
{
// Копируем следующую строку в видеобуфер
for (x=0; x<clip_width; x++)
(
// Проверка пикселя на "прозрачность" (то есть на 0);
// если пиксель "непрозрачный" - рисуем
if ((data=work_sprite[work_offset+x+x_off]))
double_buffer[offset+x+x_off] = data;
} // конец цикла по X
// Переход к следующей строке в видеобуфере
// и в растровом буфере спрайта
offset      += SCREEN_WIDTH;
work_offset += sprite_width;
} // конец цикла по У
} // конец функции
```

Как и к двум первым листингам этой главы, к Листингу 7.3 надо относиться достаточно скептически. Эта программа, конечно, делает то, что ей и положено, однако не учитывает при этом контекста. А контекст — это все! Например, в вашей игре все картинки могут располагаться посередине. Тогда в отсечении у вас просто отпадет необходимость. Возможно, ваш объект всегда будет двигаться по горизонтали, в этом случае не потребуется выполнять вертикальное отсечение и вы сможете провести оптимизацию части предварительной обработки. Иными словами, всегда пытайтесь создавать ваши функции для конкретной ситуации и для определенной игры. В данном случае передо мной стояла несколько иная задача — объяснить вам суть проблемы. Если вы поняли идею, то, потратив определенное время и силы, найдете свое правильное решение.

Контроль столкновения спрайтов

Часто бывает нужно проверить, не произошло ли в нашей игре столкновение двух или более спрайтов. Контроль столкновения между спрайтами не сложнее контроля столкновений между астероидом и кораблем. Эту тему мы обсуждали в четвертой главе. Чтобы выяснить это, достаточно посмотреть, имеются ли общие точки хотя бы у одной из четырех границ спрайта с другой областью. Алгоритм 7.2 исполняет подобный тест.

Алгоритм 7.2. Тест столкновения спрайтов.

```
// Пусть оба спрайта имеют размер width к height и предположим,
// что один из них расположен в координатах (x1,y1), а другой
// в точке (x2,y2)
if (x1>x2 and x1<x2+width and y1>y2 and y1<y2+height)
{
есть столкновение
} else
нет столкновения
}
```

Алгоритм 7.2 интересен тем, что он выполняет тест, проверяя пересечения. Мы могли бы попробовать оптимизировать его, тестируя обратное утверждение, что объекты не пересекаются. (Часто оказывается легче проверить отсутствие чего-либо, чем наличие... просто это еще одна идея

в вашу копилку.)

Дублирующая буферизация

Мы уже вкратце познакомились с понятием дублирующей буферизации в пятой главе, «Секреты VGA-карт». Теперь я хочу познакомить вас с деталями, и на это у меля есть причина.

Дублирующая буферизация используется для уменьшения мерцания экрана, вызванного перерисовкой пикселей. Дублирующий буфер — это область памяти вне видеобуфера, которая используется для построения изображения. Затем, когда изображение построено, и рисунок полностью готов к визуализации, высокоскоростной фрагмент кода перебрасывает его из дублирующего буфера в видеопамять. Обычно для этого используется команда процессора STOSW, которая перемещает несколько слов за один раз. Это наиболее эффективный способ доступа к видеобуферу.

Возможно, вам захочется узнать, работает ли дублирующий буфер медленнее чем видеобуфер? Информация, приведенная ниже, поможет вам ответить на этот вопрос:

- Во-первых, видеопамять работает крайне медленно. Операции записи и считывания осуществляются в 2-10 раз медленнее, чем при работе с оперативной памятью;
- Во-вторых, построив экранное изображение в дублирующем буфере, мы можем переместить все его 64000 байт за один прием с помощью команды STOSW;
- Наконец, если мы будем выводить все изображения непосредственно в видеобуфер, то сможем увидеть на экране весь спектр мерцаний и искажений.

Подытоживая все сказанное, можно сделать вывод, что обращение к дублирующему буферу обычно происходит быстрее, чем прямой доступ к видеопамяти, причем и анимация, и графика будут выглядеть на порядок более плавными. Единственный недостаток, это то, что дублирующему буферу требуется 64000 байт памяти, но он того стоит.

С другой стороны, обычно нет необходимости буферизировать весь экран. Если окно игрового пространства имеет только 100 пикселей в высоту, то для дублирующего буфера потребуется зарезервировать лишь 100 рядов по 320 пикселей, что составит 32K. В четырнадцатой главе в качестве примера игр для нескольких участников, связанных между собой по модему, мы создадим игру, Net-Tank. В ней мы используем дублирующий буфер только для первых 176 строк дисплея, так как остальную часть экрана будет занимать статичное, один-единственный раз созданное изображение.

Теперь, когда мы знаем, что такое дублирующий буфер и как его использовать, давайте рассмотрим некоторые примеры того, как он создается и как он действует. Модифицируем и расширим исходные программы двухмерной графики из пятой главы, «Секреты VGA-карт», использовав вместо видеопамяти дублирующий буфер (см. Листинг 7.4). Для этого в каждой функции заменим video_buffer на double_buffer. Программа CIRCLES.C делает следующее:

- Резервирует память под двойной буфер;
- Рисует в нем 1000 окружностей;
- Копирует содержимое буфера на экран за один раз.

Листинг 7.4. Программа CIRCLES.C.

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ///////////////////////////////////
#include <stdio.h>
#include <math.h>
#include <graph.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
// ОПРЕДЕЛЕНИЯ /////////////////////////////////
#define SCREEN_WIDTH      (unsigned int)320
#define SCREEN_HEIGHT     (unsigned int}200
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ ///////////////////////////////////////
unsigned char far *video_buffer = (char far *)0xA0000000L;// указатель на видеобуфер
unsigned char far *double_buffer = NULL;
// ФУНКЦИИ ////////////////////////////////////////////////////////
void Init_Double_Buffer(void)
{
double_buffer=(char far *)_fmalloc(SCREEN_WIDTH*SCREEN_HEIGHT+1) ;
_fmemset(double_buffer, 0, SCREEK_WIDTH*SCREEN_HEIGHT+l) ;
} // конец Init_Double_Buffer
//////////////////////////////////////////////////////////
void Show_Double_Buffer(char far *buffer)
{
// копирование дублирующего буфера в видеобуфер
asm {
push ds                 // сохранение регистра сегмента данных
les di, video_buffer    // вывод в видеобуфер...
```

```
lds si, buffer          // ...из дублирующего буфера
mov cx,320*200/2        // количество перемещаемых слов
cld
rep movsw               // перемещение
pop ds                  // восстановление регистра сегмента данных
}
} // конец Show_Double_Buffer
////////////////////////////////////////////////////////
void Plot_Pixel_Fast_D(int x,int y,unsigned char color)
{
// вывод,пикселей в дублирующий буфер
// используется тот факт, что 320*y = 256*y + 64*y = y<<8 + y<<6
double_buffer[((y<<8) + (y<<6)) + x] = color;
} // конец Plot_Pixel_Fast_D
////////////////////////////////////////////////////////
void Circles(void)
{
// Эта функция рисует 1000 окружностей в дублирующем буфере
// В реальной игре мы никогда не стали бы использовать столь
// нелепый алгоритм рисования окружности, а применили бы что-нибудь
// вроде таблицы выбора или другой эффективный способ. Здесь же нам
// просто надо хоть что-то нарисовать в дублирующем буфере.
int index,xo,yo,radius,x,y,color,ang;
// рисуем 1000 окружностей в случайной позиции,
// случайного цвета и размера
for (index=0; index<1000; index++)
{
// получаем параметры для следующей окружности
xo     = 20 + rand()%300;
yo     = 20 + rand()%180;
radius = 1 + rand()%20;
color  = rand()%256i
for (ang=0; ang<360; ang++)
{
x = xo + cos(ang*3.14/180} * radius;
y = yo + sin(ang*3.14/180} * radius;
Plot_Pixel_Fast_D(x,y,(unsigned char)color};
} // конец внутреннего цикла
} // конец внешнего цикла
} // конец Circles
// ОСНОВНАЯ ПРОГРАММА ////////////////////////////////////
void main(void) {
// установка видеорежима 320x200x256
_setvideomode(_MRES256COLOR) ;
// создание дублирующего буфера и очистка его
Init_Double_Buffer() ;
_settextposition (0, 0) ;
printf("Drawing 1000 circles to double buffer. \nPlease wait...");
// построение окружностей в дублирующем буфере
Circles () ;
printf("Done, press any key.");
//ожидание нажатия клавиши, прежде,чем перебросить
// окружности на экран
getch();
ShowDoubleBuffer(double_buffer);
_settextposition(0,0) ;
printf("That was quick. Hit any key to exit.");
// ожидание нажатия клавиши
getch ();
// восстановление первоначального видеорежима
_setvideomode(_DEFAULTMODE);
} // конец функции main
```

Конечно, эта программа не самая захватывающая в мире, но она дает четкое представление о том, что нас интересует. Обратите внимание, что функция вывода пикселя записывает его вместо видеобуфера в дублирующий буфер.

Использование дублирующего буфера практически убирает мерцание экрана. Однако мы не можем гарантировать, что копирование данных в видеобуфер не начнется в момент обращения

аппаратуры к экранной области памяти, и в этом случае на экране могут возникнуть искажения. В следующем разделе обсуждается то, как этого можно избежать.

Использование сигнала вертикальной синхронизации

Как вы знаете, образ на дисплее создается с помощью электронного луча, который рисует пиксель за пикселем до тех пор, пока не достигнет нижней границы экрана. Затем он возвращается по диагонали обратно к стартовой позиции и все повторяется с начала. В течение периода обратного вертикального хода луча память экрана недоступна аппаратной части, и если мы внесем изменения в видеобуфер, то не увидим их до следующего кадра.

Нам это и нужно. Период обратного вертикального хода луча — самое подходящее время для обновления видеобуфера, так как в этот момент у аппаратной части нет к нему доступа. Этот период инициируется с помощью сигнала вертикальной синхронизации (vsync), посылаемого картой VGA. Мы должны определить этот импульс и использовать период, следующий сразу за ним, чтобы паша мультипликация в точности синхронизировалась с частотой обновления экрана.

Чтобы достигнуть этого, мы должны разобраться, как определять начало периода обратного вертикального хода луча, и затем использовать наши знания для того, чтобы синхронизировать время обновления нашей графики с тем моментом, когда аппаратная часть временно забывает о существовании видеобуфера.

К счастью, карты VGA имеют специальный регистр, который отражает обратный ход луча. Он называется регистром состояния VGA и его можно прочесть, обратившись к порту 0x3DA. Из 8 битов, содержащихся в этом регистре, нас интересует четвертый справа бит (3d):

- Когда бит установлен в 1, происходит обратный вертикальный ход луча;
- Когда бит сброшен в 0, происходит перерисовка экрана.

Теперь мы можем, проверив этот бит, определить, происходит ли в данный момент обратный вертикальный ход луча. Однако если этот процесс идет, мы не можем точно узнать, находится ли он в середине, начале или конце цикла. Поэтому в таком случае мы должны дождаться его конца. Сделав это, мы будем уверены, что следующее изменение бита отметит начало цикла. Итак, мы можем написать подпрограмму, которая будет запрашивать статус бита и отслеживать начало процесса обратного вертикального хода луча. После этого она вернет управление основной программе, которая будет знать, что следующую 1/70-ю долю секунды для VGA (и 1/60-ю секунды для EGA), можно совершенно спокойно писать в видеобуфер.

Я написал демонстрационную программу VSYNC.C, которая выполняет подобную синхронизацию и вычисляет количество периодов обратного вертикального хода луча как функцию от времени. Так как за секунду выполняется 70 циклов регенерации экрана, то после минутной работы программы результат должен оказаться равным 4200. Листинг 7.5. содержит исходный текст этой программы.

Листинг 7.5. Подсчет количества циклов обратного вертикального хода луча (VSYNC.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ////////////////////////////////////////
#include <dos.h>
#include <bios.h>
#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <graph.h>
// ОПРЕДЕЛЕНИЯ /////////////////////////
#define VGA_INPUT_STATUS_1   0x3DA // регистр состояния VGA,
// бит 3 - сигнал вертикальной синхронизации
// 1 - происходит обратный вертикальный ход луча
// 0 - нет обратного вертикального хода луча
#define VGA_VSYNC_MASK 0x08
// маскировка неиспользуемых битов регистра
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ ////////////////////
unsigned char far *video_buffer = (char far *)0xA0000000L;// указатель на видеопамять
// ФУНКЦИИ ///////////////////////////////////////////////
void Wait_For_Vsync(void)
{
// эта функция ожидает начала обратного вертикального хода луча, а
// если луч уже совершает обратный ход - ожидает следующего цикла
while (_inp(VGA_INPUT_STATUS_1) & VGA_VSYNC_MASK)
{
// обратный вертикальный ход луча - ничего не делаем
} // конец оператора while
// ожидаем прихода сигнала vsync и возвращаем
// управление вызвавшей функции
while (! (_inp(VGA_INPUT_STATUS_l) & VGA_VSYNC_MASK) )
{
// ожидание начала обратного вертикального
// хода луча - ничего не делаем
```

```
} // конец оператора while
// начался обратный вертикальный ход луча —
// возвращаем управление вызвавшей функции
} // конец Wait_For_Vsync
// ОСНОВНАЯ ПРОГРАММА ////////////////////////////////////  .
void main(void)
{
long nuinber_vsyncs=0;  // хранит количество циклов
// обратного вертикального хода луча
while(!kbhit())
{
// ожидание vsync
Wait_For_Vsync(};
// выводим графику или что-либо еще пока происходит
// обратный вертикальный ход луча и мы имеем в своем
// распоряжении 1/70 секунды! Обычно в это время выполняется
// копирование дублирующего буфера в видеопамять
// увеличивается количество циклов vsyncs
number_vsyncs++;
// вывод на экран
_settextposition(0,0) ;
printf ("Number of vsync's = %ld   ",number_vsyncs);
} // конец оператора while
} // конец функции main
```

Существует миллион вещей, для которых можно использовать обратный вертикальный ход луча:

- Мы можем использовать это время в качестве основы для некоторых процессов или событий;
- Мы можем использовать это для того, чтобы игровой цикл выполнялся с частотой 70Гц на любой машине;
- Наконец, мы можем использовать это для синхронизации процесса обновления графики с периодом обратного вертикального хода луча.

Замечание

Я хотел бы сказать вам еще и о том, чего вы делать не должны. Речь идет об использовании прерывания по обратному вертикальному ходу луча. Персональные ...компьютеры имеют аппаратную поддержку вектора прерывания периода обратного вертикального хода луча. Однако такая поддержка не является стандартной для IBM-совместимых персональных компьютеров и не всегда может работать. К тому же, при этом используется INT2, который зарезервирован IBM для фатальных ошибок аппаратного обеспечения. Поэтому я не буду говорить об этом прерывании, так как оно зависит от марки компьютера и большинство программистов его не используют.

Перейдем к следующей интересной и захватывающей теме: мультипликации с помощью регистров цвета.

Мультипликация с помощью регистров цвета

Зачастую мы видим не то, что есть на самом деле. Поэтому многие эффекты и мультфильмы делаются не так, как вы могли бы подумать. Мультипликация с Помощью регистров цвета использует один из таких технических трюков, в результате которого можно очень просто получить то, что было бы чрезвычайно сложно или дорого при решении задачи «в лоб».

Мультипликация с помощью регистров цвета основана на том факте, что образ — это набор цветов. Эти цвета, в свою очередь, определены значениями RGB в таблице выбора цвета. Например, представим, что мы имеем два идентичных объекта — два изображения дерева, каждое из которых состоит из пикселей восьми оттенков зеленого. Мы создаем два набора цветов, каждый из которых содержит одинаковые восемь оттенков. Затем, используя для каждого дерева свой набор, мы рисуем две картинки: то есть изображаем первое дерево, используя первый набор цветов, и второе — используя второй, как это показано на рисунке 7.5.

Рис. 7.5. *Использование двух наборов одинаковых цветов с различными значениями регистра цвета.*

Если мы передадим изображение этих деревьев на экран, то увидим два дерева, имеющих одинаковый вид, но нарисованных с применением различных Значений регистров цвета. Теперь перейдем к самому интересному: что если мы обнулим первый набор этих регистров? Первое дерево должно будет исчезнуть.

На этом и основана мультипликация с использованием регистров цвета. Схематично это происходит так;

- Каждое из интересующих нас изображений мы выполняем с помощью различных наборов регистров цвета с одинаковыми значениями RGB;
- Затем делаем одно из изображений объекта невидимым, обнуляя соответствующие регистры цвета;
- Потом заменяем регистры цвета «включенного» объекта значениями RGB «выключенного» изображения, оставляя другой банк регистров цвета обнуленным.

Это создает ощущение движения объекта от одной позиции к другой, хотя на самом деле этого не происходит. И все это осуществляется изменением лишь нескольких регистров цвета.

Такую технику можно использовать для создания объектов, изменяющих свой внешний вид, движущихся по экрану или исчезающих. Чтобы показать вам, как это может работать, я создал демонстрационную программу BIRDANI.C, которая рисует маленькую птичку, летающую по кругу, но на самом деле неподвижную. Я нарисовал птичку в графическом редакторе, использовав при этом 13 оттенков серого цвета. Затем, с помощью программного обеспечения для работы с PCX, описанного в пятой главе, «Секреты VGA-карт», загрузил это изображение. После чего обнулил все цветовые регистры, составляющие эти 13 оттенков серого цвета. Затем я поместил серый в первый цветовой регистр, потом обнулил его, поместив серый в следующий цветовой регистр и так далее- В результате все это выглядит так, как будто птица взмахивает крыльями и летит по кругу. Листинг 7.6 содержит текст этой программы.

Замечание

Отныне в этой книге я буду использовать перекомпилированные библиотеки, поэтому вы больше не увидите старых функций в листингах программ. Текст большинства из этих функций я поместил в файлы с именем GRAPHO.C. Соответственно, файл заголовка будет называться GRAPHO.H. Чтобы функции GRAPH0.C были доступны для наших программ мы должны сначала скомпилировать сам GRAPH0.C (только убедитесь, что вы используете модель памяти MEDIUM). Затем во время липковки скомпонуйте получившийся объектный файл с той программой, с которой вы сейчас работаете. Не забудьте включить в вашу программу файл заголовка GRAPH0.H, иначе функции из GRAPH0.C будут иедоступны. Вы могли бы также создать библиотеку графических функций, используя программу Microsoft's Library Manager, LIB- Правда у нас только один файл с исходным текстом библиотечных функций, поэтому данная программа для нас излишне мощна. Однако решать, конечно же, вам. Короче говоря, если код из GRAPH0.C, GRAPH0.H и функция установки видеорежима каким-либо образом включены в программы этой главы, то они должны работать.

Листинг 7.6. Мультипликация с помощью изменения регистров цвета (BIRDANI.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ////////////////////////////////////
#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <bios.h>
#include <fcntl.h>
#include <memory.h>
#include <malloc.h>
#include <math.h>
#include <string.h>
#include "graphl.h" // включаем нашу графическую библиотеку
```

```
// ОПРЕДЕЛЕНИЯ ////////////////////////////////////////
#define BIRD_START_COLOR_REG 16
#define BIRD_END_COLOR_REG   28
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ //////////////////////////////////
unsigned int far *clock = (unsigned int far *)0x0000046C;
// указатель на внутренний таймер 18.2 "тик"/с
pcx_picture birds;
// ФУНКЦИИ ////////////////////////////////////////////////
void Timer(int clicks)
{
// эта функция использует внутренний таймер с частотой 18.2 "тик"/с.
// 32-битовое значение этого таймера находится по адресу 0000:046Ch
unsigned int now;
// получаем текущее время
now = *clock;
// Ожидаем до истечения указанного периода времени.
// Заметьте, что каждый "тик" имеет длительность-примерно в 55 мс
while(abs(*clock - now) < clicks){}
} // конец Timer
////////////////////////////////////////////////////////
void Animate_Birds(void)
{
// эта функция перемещает птичку, изображенную 13-ю различными // цветами,
последовательно включая один цвет и выключая остальные
RGB_color color_l, color_2;
int index;
// очистка каждого из цветовых регистров,
// используемых в изображении птички
color_l.red   = 0;
color_l.green = 0;
color_l.blue  = 0;
color_2.red   = 0;
color_2.green = 63;
color_2.blue  = 0;
// очистка всех цветов
for (index=BIRD_START_COLOR_REG;
index<=BIRD_END_COLOR_REG; index++)
{
Set_Palette_Register(index, (RGB_color_ptr)&color_l);
} // конец цикла for
// делаем первую птичку зеленой и затем
// последовательно меняем цвета
Set_Palette_Register(BIRD_START_COLOR_REG,
   (RGB_color_ptr)&color_2) ;
// мультипликация цветами
while(!kbhit())
{
// меняем цвета
Get_Palette_Register(BIRD_END_COLOR_REG,
(RGB_color_ptr)&color_l) ;
for (index=BIRD_END_COLOR_REG-l;
index>=BIRD_START_COLOR_REG; index—) {
Get_Palette_Register(index,(RGB_color_ptr)&color_2) ;
Set_Palette_Register(index+l,(RGB_color_ptr)&color_2);
} // конец цикла for
Set_Palette_Register(BIRD_START_COLOR_REG,
         (RGB_color_ptr)&color__l);
// небольшая пауза
Timer(3);
) // конец оператора while
} // конец Animate_Birds
// ОСНОВНАЯ ПРОГРАММА /////////////////////////////////////
void main(void)
{
int index,done=0;
// установка видеорежима 320x200x256
Set_Mode(VGA256);
```

```
// инициализация PCX-файла, который содержит изображение птички
PCX_lnit((pcx_picture_ptr)&birds) ;
// загрузка файла PCX
PCX__Load("birds.pcx", (pcx_picture_ptr)&birds,1);
PCX_Show_Buffer((pcx_picture_ptr)&birds);
PCX_Delete((pcx_picture_ptr)&birds);
_settextposition(0,0);
printf("Hit any key to see animation."};
getch()
_settextposition(0,0) ;
printf("Hit any key to Exit.          ");
Animate_Birds();
// возврат в текстовый режим
Set_Mode(TEXT_MODE);
} // конец функции xnain
```

Конечно, программа BIRDANI.C элементарна. Однако с помощью этой техники можно получить потрясающие эффекты. Один из них мы и обсудим в следующем разделе.

Освещение ваших игр

Освещение, тени и другие эффекты традиционно выполняются с помощью сложной математики, основанной на физических моделях. При построении этих моделей принимаются во внимание такие факторы как угол падения, расстояниеИ до источника света, тип источника света, тип материала, плотность атмосферы и т. д. (более подробно об этом рассказывается в шестой главе, «Третье измерение». Не удивительно, что для получения фотографически точных изображений тратятся часы или даже дни работы суперкомпьютеров типа Cray XMP! Уравнения, которые надо для этого решить, да и количество операций, которое необходимо выполнить — все это слишком грандиозно для нашего маленького персонального компьютера. Однако мы попытаемся, используя смекалку и упрощенные модели освещения, реализовать несколько «достаточно хороших» для наших потребностей эффектов. Потрясающие световые эффекты знаменитой игры DOOM впечатляют, по, тем не менее, в них нет ничего невозможного. Как говорится, глаза боятся, а руки делают.

В качестве примера представим, что мы нарисовали комнату, используя регистры цвета от 1 до 32. Эти регистры содержат оттенки серого (вместе с оттенками красного, которыми изображены ругательства, написанные на стенах). Что произойдет, если мы постепенно увеличим значение RGB в каждом из этих регистров? Вы увидите, что комната стала светлее. Если же вы уменьшите каждое значение RGB во всех регистрах цвета, в комнате станет темнее.

Конечно, вы должны быть уверены в том, что сохранится процентное соотношение компонентов. Иначе цвет может измениться вместе с интенсивностью. Другими словами, вы должны увеличивать или уменьшать величины, составляющих каждого регистра цвета пропорционально их исходному значению. Следовательно, чтобы уменьшить интенсивность определенного регистра цвета, скажем, 26, вы должны вначале выяснить, в каком процентном соотношении он содержит красный, зеленый и синий цвета. Затем вам надо вычислить значение, которое должно быть добавлено или отнято от каждой составляющей для того, чтобы исходный цветовой баланс не изменился.

Для примера, давайте будем уменьшать каждое значение RGB на 5 процентов за каждую итерацию. Чтобы сделать это, мы должны вычислить 5 процентов для каждого из значений RGB и вычесть их из каждого компонента. Таблица 7.2 показывает результаты этой процедуры за небольшой промежуток времени для некоторого произвольного цвета RGB.

Таблица 7.2. Уменьшение интенсивности цвета при сохранении цветового баланса.

(Цветовой баланс с использованием красного цвета в качестве эталона)

	R	G	B	Цветовой баланс
Итерация 1	60	40	20	1:0.66:0.30
Итерация 2	57	38	19	1:0.66:0.33
Итерация 3	54	36	19	1:0.660.33

Как вы можете видеть, если пошагово уменьшать каждую из составляющих на пять процентов, цветовой баланс будет сохраняться. Конечно, вычисление пяти процентов каждого

значения при каждой итерации слишком дорого. Чтобы избежать этого, мы можем попробовать просто вычитать постоянное значение из каждого компонента регистра цвета, игнорируя цветовой баланс. Однако когда интенсивность цвета существенно увеличится или уменьшится, станет явно заметным и изменение оттенка исходного цвета.

Компромиссным вариантом может стать подсчет трех констант на основании исходного цветового баланса, а затем их прибавление или вычитание. Это уменьшит ошибку, вносимую в цветовой баланс. В нашем случае мы можем использовать 5 для красного компонента, 0.66x5 - для зеленого и 0.3x5 - для синего. Это дает приемлемые результаты. Используя этот метод, мы получим следующие значения RGB после первой итерации:

R	G	B	Цветовой баланс
55	37	19	1:0.67:0.34

Ошибка» возникающая при расчете таким методом, пренебрежительно мала, и если уменьшение или увеличение компонентов RGB будет производиться всего несколько раз, изменение тона исходного цвета будет практически незаметным.

Связь мультипликации с контекстом

Как это ни удивительно, но на связь мультипликации с контекстом крайне редко обращают внимание в компьютерных играх. Контекст означает «связь с окружением или основой». В играх контекстом являются разнообразные дейст вия по ходу игры. Что бы ни делал персонаж в игре: умирал, прыгал, стрелял или еще что-нибудь, -- все это часть контекста игры. Мультипликация и различные эффекты, которые связаны с контекстом, придают игре дополнительную реалистичность и размах.

Попробую пояснить на примере, что я имею в виду. Представьте, что мы имеем игру, в которой герой может гулять, прыгать, бегать и стрелять. Независимо от того, перепрыгивает ли он через паука или озеро, во время прыжка мультипликация обычно бывает всегда одна и та же. Но что если для каждого случая, или соответствующего контекста мы сделаем отдельный мультик? Например, когда герой перепрыгивает просто через лужу, то он не совершает ничего кроме собственно прыжка. А вот когда ему приходится перепрыгивать через паука, то он может, например, издать пронзительный крик ужаса и показать изумительную сноровку в прыжках. (И действительно, приземление на большого склизкого паука, возможно, не самый лучший опыт для маленького героя, которого заманили в виртуальный компьютерный мир.)

Итак, контекстнозависимая мультипликация, используя набор соответствующих вариантов движений, учитывает тем самым окружение и внешние обстоятельства в каждый момент игрового действия, что дополнительно разнообразит зрительное восприятие игры.

«Animotion»

*Animation (animation (*мультипликация *)+ motion (движение))* — это придуманное мною слово, которое должно войти в словарь терминов компьютерных игр. (Дальше я тоже буду выдумывать слова, но они уже не будут так хороши как это.) Оно описывает точное слияние анимации (мультипликации) и движения.

В игре Tombstone четвертой главы, "Механизмы двухмерной графики", маленький ковбой гуляет по улице. В действительности это больше похоже на то, что он одновременно ковыляет и совершает дикие прыжки (если такое вообще можно представить). Вся проблема заключается в принципе его движения, которое не было синхронизировано с мультипликационными кадрами. Многие объекты, в компьютерных играх имеют *постоянную* скорость. Такие игровые объекты как ракета и прочие летающие предметы — хороший тому пример, Однако к имеющим под собой опору (например, землю) объектам, которые должны выглядеть реально, нужно применять совсем другой подход.

«Animotion» абсолютно необходим, если вы хотите, чтобы движение -ходьба, бег или прыжки — выглядели реалистично. Иначе мультипликационные объекты выглядят неестественно. Мы не можем просто в цикле менять мульти пликационный кадр и одновременно передвигать объект на произвольное расстояние. Мы должны рассмотреть каждый кадр и определить, на сколько *в действительности* следует перемещать объект для этого кадра. Затем мы создаем таблицу выбора для движения, в которой в качестве индекса используем номера кадров. В этой таблице будут содержаться величины перемещений, которые и будут использоваться при оживлении картинки.

На рисунке 7.6 изображены мультипликационные кадры движения худого человечка, которого-мы назовем «человечек-палка».

(У «человечка-палки» серьезные проблемы со зрением — он имеет только один глаз.) Для движения человечка у нас будет 12 мультипликационных кадров. Для каждого из кадров я прикинул и задал величину перемещения, при котором движение становится действительно похожим на прогулку. Немного помучавшись, я получил для каждого мультипликационного кадра значения, приведенные в таблице 7.3.

Таблица 7.3. Таблица перемещений.

Кадр	Значение	Кадр	Значение	Кадр	Значение
1	17	5	3	9	6
2	0	6	0	10	2
3	6	7	17	11	3
4	2	8	0	12	0

Чтобы показать на примере, насколько «animotion» превосходит по качеству постоянное движение, я создал программу в которой игрок, нажимая на клавишу пробел, может выбирать между ними. Нашим «подопытным кроликом» снова будет беззащитный «человечек-палка». Мы собираемся поместить его в действительно ужасный городской квартал и при этом у него не будет ничего, кроме пары ног, пары рук и пары глаз. (Ой, извините! Глаз-то всего один.) Теперь посмотрите, как он разгуливает по экрану. Затем нажмите клавишу пробела, чтобы перейти с постоянного движения к «animotion». Вы увидите большую разницу. Текст программы приведен в Листинге 7.7, она также использует функции из GRAPH0.C, поэтому не забудьте скомпоновать программу с этим модулем.

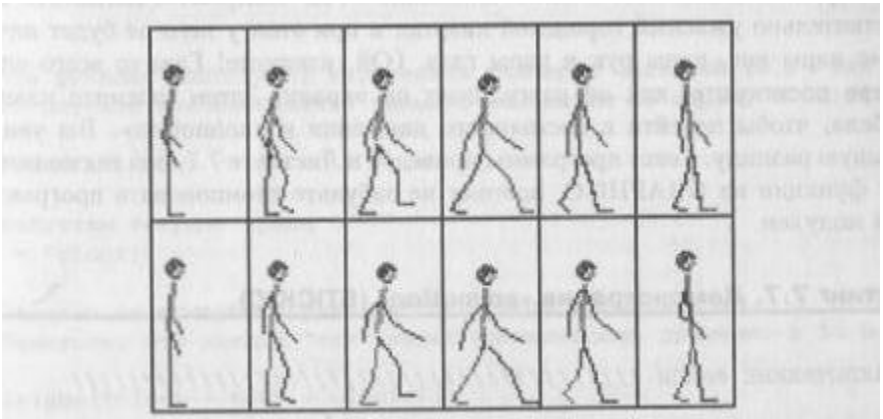


Рис. 7.6. Мультипликационные кадры для игрушечного человечка.

Листинг 7.7. Демонстрация «animotion» (STICK.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ////////////////////////////////////////
#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <bios.h>
#include <fcntl.h>
#include <memory.h>
#include <malloc.h>
#include <math.h>
#include <string.h>
#include "graph0.h" // включаем нашу графическую библиотеку
// ОПРЕДЕЛЕНИЯ /////////////////////////////////////////////
#define VEL_CONST -1  // флаг постоянной скорости перемещения
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ ///////////////////////////////////
unsigned int far *clock = (unsigned int far *)0x0000046C;
// указатель на внутренний таймер 18.2 "тик"/с
sprite object;
pcx_picture stick_cells, street_cells;
// таблица выбора содержит величины перемещения для каждого
// мультипликационного кадра, чтобы движение было более реалистичным
int object_vel = {17,0,6,2,3,0,17,0,6,2,3,0};
// ФУНКЦИИ /////////////////////////////////
```

```
void Timer (int clicks)
{
// эта функция использует внутренний таймер с частотой 18.2 "тик"/с
// 32-битовое значение этого таймера находится по адресу 0000:046Ch
unsigned int now;
// получаем текущее время
now = *clock;
//Ожидаем до истечения указанного периода времени.
// Заметьте, что каждый "тик"' имеет длительность примерно в 55 мс
while(abs(*clock - now) < clicks)() {}
// конец Timer
// ОСНОВНАЯ ПРОГРАММА ////////////////////////////////////
void main(void) {
int index, done=0,
vel_state=VEL_CONST;
// установка видеорежима 320x200x256
Set_Mode(VGA256);
// установка размера для системы отображения спрайта
sprite_width = 32;
sprite_height =64;
// инициализация файла PCX, который содержит изображение улицы
PCX_Init((pcx_picture_ptr)&street_cells) ;
// загрузка файла PCX, который содержит изображение улицы
PCX__Load("street.pcx", (pcx_picture_ptr)&street_cells,1} ;
PCX_Show_Buffer((pcx_picture_ptr)&street_cells) ;
// используем буфер PCX как дублирующий
double_buffer = street_cells.buffer;
Sprite_Init((sprite_ptr)&object,0,0,0,0, 0,0);
// инициализация файла PCX, который содержит кадры спрайта
PCX_Init((pcx_picture_ptr)&stick_cells);
// загрузка файла PCX, который содержит кадры спрайта
PCX_Load("stickman.pcx", (pcx_picture_ptr) &stick_cells,1) ;
// выбираем 6 кадров движения
PCX_Grap_Bitmap((pcx_picture_ptr)&stick_cells,
(sprite_ptr)&object,0,0,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&stick_cells,
(sprite_ptr)&object,1,1,0);
PCX_Grap_Bitmap( (pcx_picture_ptr) &stick_cells,
(Sprite_ptr)&object,2,2,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&stick_cells,
(sprite_ptr)&object,3,3,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&stick cells,
(sprite_ptr)&object/4,4,0);
PCX_Grap_Bitmap( (pcx_picture_ptr) &stick_cells, (sprite_ptr) &object, 5, 5, 0);
PCX_Grap_Bitniap ( (pcx_picture_ptr) &stick_cells,(sprite_ptr)&object,6, 0,1);
PCX_Grap_Bitmap((pcx_picture_ptr)&stick_cells,
(sprite_ptr)&object,7, 1,1);
PCX_Grap_Bitmap ( (pcx_picture_ptr) &stick_cells,
(sprite_ptr)&object,8, 2,1) ;
PCX_Grap_Bitmap((pcx_picture_ptr)&stick_cells,
(sprite_ptr) &object, 9, 3,1);
PCX_Grap_Bitmap((pcx_picture_ptr)&stick_cells,(sprite_ptr)&object,10,4,1);
PCX_Grap_Bitmap((pcx_picture_ptr)&stick_cells,
(sprite_ptr)&object,11,5,1);
// файл stickman.pcx больше не нужен
PCX_Delete((pcx_picture_ptr)&stick_cells);
// настраиваем параметры человечка
object.x        = 10;
object.y        = 120;
object.curr_frame = 0;
// сохраняем фон
Behind_Sprite((sprite_ptr)&object);

// главный цикл
while(!done)
{
// стираем спрайт
```

```
Erase_Sprite((sprite_ptr)&object) ;
// увеличиваем номер кадра на единицу
if (++object.curr_frame > 11) object.curr_fcame = 0;
// перемещаем спрайт, используя или постоянную скорость,
// или таблицу выбора
if (vel_state==VEL_CONST)
{
object.x+=4;
}
// конец if
else
/         {
// используем номер кадра для определения величины
// перемещения по таблице выбора
object.x += object_vel[object.curr_frame];
} // конец else
}
```

Вы наверняка обратили внимание на горы и местность за окнами, которые перемещаются при ваших поворотах. Фактически, этот пейзаж есть не что иное, как плоское изображение, прокручиваемое как фон. Эти изображения, чтобы они выглядели трехмерными, прорисовываются специальным образом, но, тем не менее, в основе их перемещения лежит все та же прокрутка двухмерного изображения.

Персональные компьютеры могут иметь специальное аппаратное обеспечение, облегчающее прокрутку. Некоторые карты VGA имеют до мегабайта оперативной памяти, что дает возможность рисовать мир вашей игры прямо в видеобуфере, поручая производить прокрутку самой видеокарте. Однако существует две проблемы:

Во-первых, если мы будем полагаться на определенную аппаратуру, наши программы станут аппаратнозависимыми и не будут работать на картах VGA с меньшей памятью;

Во-вторых, прокрутка в режиме 13h существенно сложнее, чем прокрутка в режимах EGA, так как видеопамять в этом случае не разбита на несколько плоскостей. Эта означает, что мы не можем использовать в данном режиме аппаратную прокрутку, таким же образом как в режимах EGA.

Таким образом, мы не будем использовать для прокрутки аппаратное обеспечение персонального компьютера. Мы будем это делать программно, применяя блочное копирование из памяти в видеобуфер. Используя этот метод, мы будем абсолютно независимыми от аппаратного обеспечения, следовательно наши программы будут более гибкими.

Прокрутку целого экрана или его части можно осуществлять двумя путями:

- Можно нарисовать все пространство игры в обширном буфере памяти. Однако шесть предварительно нарисованных экранов займут 6x64000 байтов, то есть 384K. Это довольно большие потери памяти. В любом случае, как мы только что говорили, мы должны будем сделать воображаемое окно, перемещающееся по этому буферу и отображать все, что в него попадает, на экран;
- Второй метод более медленный, но требует меньше памяти. Он основан на генерации изображения «на лету». Под выражением "на лету" я подразумеваю, что мир будет представляться с помощью иначе структурированных данных - например, в виде двухмерной матрицы, где каждая ячейка 8x8 пикселей ставится в соответствие растровому изображению части игрового пространства. Такой ячеистый мир в шесть экранов будет занимать всего 6000 байт. При перемещении окна по ячейкам матрицы, на экране будут воспроизводиться соответствующие растровые изображения.

Главный недостаток этого метода — скорость. Ведь для каждой ячейки нужно будет найти соответствующее растровое изображение и перенести его в дублирующий буфер или на экран, поэтому на перерисовку экрана будет уходить много времени. Вы наверняка встречали игры, которые выглядят не очень симпатично во время прокрутки из-за того, что перерисовка экрана явственно заметна. Это как раз и происходит из-за применения второго метода. Мы должны принести в жертву либо время, либо память. Выбор как всегда за вами.

Чтобы показать вам пример скроллинга, я написал программу, которая создает игровое пространство размером 640x100 пикселей. Я двигаю окно по изображению мира игры и передаю его содержимое в середину экрана. В этой игре мир состоит из звезд и гористого горизонта. (Этот пейзаж немного напоминает игру Defender.) Перемещая с помощью клавиатуры окно просмотра вправо и влево, вы можете прокрутить весь пейзаж. Листинг 7.8 содержит текст этой программы, которая называется DEFEND.C.

Листинг 7.8. Пример прокрутки (DEFEND.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ////////////////////////////////////////
#include <stdio.h>
#include <math.h>
#include <graph.h>
#inciude <malloc.h>
#include <memory.h>
```

```
#include <string.h>
// ОПРЕДЕЛЕНИЯ ////////////////////////////////////////
#define SCREEN_WIDTH      (unsigned int)320
#define SCREEN_HEIGHT     (unsigned int)200
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ //////////////////////////////////////
unsigned char far *video_buffer = (char far *)0xA0000000L;//указатель на видеобуфер
unsigned char far *double_buffer = NULL;
// ФУНКЦИИ ////////////////////////////////////////////////
void Show_View_Port(char far *buffer,int pos)
{
// Копирование части дублирующего буфера на экран
unsigned int y,double_off, screen_off;
// нужно переместить 100 строк, перемещаем их построчно
for (y=0; y<100; y++) {
// расчет начального смещения дублирующего буфера
//y * 640 +pos
double_off = ((y<<9) + (y<<7) + pos);
// расчет начального смещения в видеобуфере
// y * 320 + 80
screen_off = (((y+50)<<8) + ((y+50)<<6) + 80);
// перемещение данных
_fmemmove ((char far *)&video_buffer[screen off],
(char far *)&double_buffer[double_off],160);
} // конец цикла for
} // конец Show
View_Port ////////////////////////////////////////////////////////////
void Plot_Pixel_Fast_D2(int x,int y,unsigned char color)
{
// прорисовка пикселей в дублирующем буфере нашего виртуального
// экрана размером 640x100 пикселей
// учтем, что 640*y = 512*y + 128*y = y<<9 + y<<7
double_buffer[ ((y<<9) + (y<<7)) + x] = color;
} // конец
Plot_Pixel_Fast_D2 ////////////////////////////////////////////////////////
void Draw_Terrain(void) {
// эта функция рисует ландшафт в дублирующем буфере
// размером 640x100 пикселей
int x,y=70,index;
// очистка памяти
_fmemset(double_buffer,0,(unsigned int}640*(unsigned int)100);
// рисуем звезды
for (index=0; index<200; index++)
{
Plot_Pixel_Fast_D2(rand()%640,rand()%70,15);
} // конец цикла for
//рисуем горы
for (x=0; x<640; x++)
{
// расчет смещения
y+=-1 + rand()%3;
// проверяем, находятся ли горы в приемлемых границах
if (y>90) y=90;
else
if (y<40) y=40;
// рисуем точку в дублирующем буфере
Plot_Pixel_Fast_D2 (x,y, 10);
} // конец цикла for } // конец Draw_Terrain
// ОСНОВНАЯ ПРОГРАММА ////////////////////////////////////
void main(void)
{
int done=0,sx=0;
// установка видеорежима 320x256x256
_setvideomode(_MRES256COLOR);
_settextposition(0,0);
printf("Use < > to move. Press Q to quit.");
// рисуем небольшое окно
_setcolor(l);
```

```
_rectangle(_GBORDER, 80-1,50-1,240+1,150+1) ;
// резервируем память под дублирующий буфер
double_buffer=(char far *)_fmalloc(SCREEN_WIDTH*SCREEN_HEIGHT+1);
Draw_Terrain() ;
Show_View_Port(double_buffer,sx) ;
// главный цикл
while (!done)
{// не Нажал ли игрок клавишу?
if (kbhit())
switch(getch())
{
case ',': // переместить окно влево, если это возможное
{
sx-=2;
if (sx<0)
sx=0;
} break;
case '.':// переместить окно вправо если это возможно
{
sx+=2;
if (sx > 640-160)
sx=640-160;
} break;
case 'q': // игроку надоело?
{
done=1; } break;
} // конец оператора switch
// копируем окно просмотра на экран
Show_View_Port(double_buffer,sx);
_settextposition(24,0);
printf("Viewport position = %d ",sx);
} // конец оператора if
} // конец оператора while
// восстановление видеорежима
_setvideomode(_DEFAULTMODE);
}// конец функции main
```

Если вы похожи на меня, то, вероятно, захотите написать игру типа Defender. Отлично! Отложите книгу — и вперед. Не беспокойтесь, я подожду. По крайней мере, попытайтесь в своей программе заставить джойстик управлять полетом корабля.

О, вы уже вернулись? Тогда поговорим о тех специальных потрясающих эффектах, без которых ни одна игра никогда не завоюет популярности.

Специальные эффекты

Под *специальными* эффектами обычно понимают такие зрительные эффекты, которые, выходят за рамки обычного и хорошо запоминаются. Современные компьютерные игры все больше и больше походят на настоящее кино и имеют весьма впечатляющую графику. Однако всегда можно найти что-то еще, что заставит игрока удивится. Поэтому попытаемся добиться от него возгласа:

"Ого!!!"

Специальные эффекты могут быть какими угодно, начиная от ярких взрывов и кончая сверхъестественными явлениями, психоделической графикой и всем-чем-хотите. Например, сейчас я работаю над компьютерной моделью системы частиц для того, чтобы монстр разлетался на биллионы маленьких атомов, несущихся каждый по своей орбите. Это должно выглядеть отлично! Я не уверен, что я сделаю это, но если мне это удастся, то это будет сводить с ума!

Специальные эффекты — это та область, в которой вы должны полностью Проявить себя, потому что это самая творческая часть любой компьютерной игры. Запомните одну вещь: в любой игре должен быть по-крайней мере один потрясающий воображение эффект, такой эффект, который игрок не скоро забудет! Итак, к делу, тем более что в этом разделе мы поговорим и о других важных вещах.

Фоновая мультипликация

Мы просто не можем пройти мимо фоновой мультипликации. Компьютерные Игры в плане эффектов обошли своих предшественников — игры с фишками. Добавление небольших мультиков и эффектов, которые впрямую не связаны с Действиями в игре делает компьютерные игры забавнее и оригинальнее.

Хороший пример этому — оформление в игре Karate. Летающие птички, время от времени садящиеся на вершину действующего вулкана, — это как раз то, что надо. Для игры, созданием которой мы займемся в конце этой книги, я Придумал светлячков, которые просто мигают то тут,

то там. Они не служат для Какой-то определенной цели, а просто делают мир игры немного более реалистичным и полным.

Эти небольшие, длящиеся не более минуты эпизоды вы должны добавить Уже после того, как игра сделана. Всегда пытайтесь добавить в оформление игры Немного небольших мультфильмов и зрительных эффектов — это придаст вашей игре своеобразие.

Экранные эффекты

Компьютерные игры становятся все больше и больше похожими на кинофильмы. В титрах количество программистов и инженеров совсем невелико. Их заменили режиссеры и продюсеры. И вообще все выглядит так, как будто это не дисплей компьютера, а кино- или телеэкран. В кино и по телевизору мы видим изображения смикшированные и растушеванные, кроме того, там применяются и другие эффекты (например, изменяется план или фокус съемки). В компьютерных играх также используют разнообразные экранные эффекты. Например, прохождение вами очередного уровня игры может отмечаться такими эффектами, как гаснущий или тающий экран (или еще что-нибудь эдакое)!

Смена экранов влияет на впечатление от игры. Экран, на котором просто меняется фон эпизодов, быстро надоедает. Я бы предпочел, например, чтобы новая картинка появлялась из-под разлетающихся осколков старого изображения или что-нибудь еще более замысловатое.

Программировать эффектные способы смены экрана на самом деле достаточно просто. Большинство из алгоритмов требует всего нескольких строк кода, а самые сложные из них не более дюжины строк. Это еще одна область, в которой вы должны будете проявить все свои таланты, чтобы добиться действительно стоящих результатов. Для примера я объединил три различных варианта исчезновения содержимого экрана в программу SCREENFX.C. Текст этой программы приведен в Листинге 7.9. Она использует функции работы с изображениями в формате PCX (созданные в пятой главе, «Секреты VGA-карт») и некоторые функции из библиотеки GRAPH0.C.

Программа SCREENFX.C демонстрирует три эффекта:

- Гаснущее изображение;
- Исчезновение изображения по точкам;
- Оплывание изображения.

Попытаемся кратко разобраться в механизмах каждого из них, чтобы понять основные идеи экранных эффектов вообще.

Гаснущее изображение

Этот эффект уже много лет пользуется популярностью у программистов, потому что получить его фантастически легко — для этого требуется всего несколько строк кода.

По существу мы медленно уменьшаем значение всех регистров цвета до тех пор, пока они не достигнут нуля. Как мы узнали в этой главе в разделе «Освещение ваших игр», лучше всего сохранять эквивалентный баланс цветов, для чего надо пропорционально увеличивать или уменьшать интенсивность каждого цвета. Однако в случае данного эффекта действие разворачивается так быстро, что цвета разобрать сложно, поэтому о цветовом балансе можно не беспокоиться.

Поэтому программа, приведенная в Листинге 7.9 просто уменьшает каждый компонент каждого цвета в пяти регистрах до тех пор, пока все три компонента — красный, зеленый и синий — не становятся равными нулю. На этом функция завершается.

Исчезновение Изображения

Этого замечательного эффекта можно достичь вообще одной строчкой. Я вам продемонстрирую ее через минуту.

Данный эффект достигается за счет образования черных дыр в пикселях изображения на экране. Все что надо сделать, это последовательно нанести на экран случайным образом тысячи черных точек. Так как экран содержит последнее изображение (комнаты, пейзажа или чего-нибудь еще), прорисовка этих черных точек приведет к тому, что создастся эффект растворения картинки в воздухе,

Здесь приведены две самые короткие строчки кода, которые мне удалось придумать:

```
for (index=0; index<=300000; index++)
Plot_Pixel_Fast(rand()%320, rand()%200, 0);
```

Этот код просто размещает 300000 черных пикселей по всему экрану. Почти наверняка такое большое количество пикселей разрушит все изображение на экране. (Мы могли бы рассчитать точное количество итераций, но это заняло бы лишнее время. Число 300000 выглядит вполне приемлемым.)

Этот эффект может быть использован для замены одного экрана другим, если в нем использовать не черные пиксели, а пиксели новой картинки, случайным образом выбирая их координаты X и Y.

Оплывание изображения

Этот эффект появился в игре DOOM и быстро распространился по другим играм. Мой вариант не столь впечатляющий. Однако, этого и следовало ожидать, ведь я потратил на него всего 15 минут.

Таяние изображения осуществляется благодаря движению сверху вниз с разной скоростью 160 маленьких «червячков» под влиянием силы тяжести. И .Когда "червячки" движутся вниз, они поедают пиксели. Через некоторое время большинство из них достигает низа и растворение заканчивается. (Я подозреваю, Что и в DOOM использовалась подобная техника. Таким образом,

каждая вертикальная лента масштабируется вместо простого стирания пикселей.)

Демонстрация смены экрана

Гаснущие, исчезающие и растворяющиеся изображения — все они выглядят неплохо. Здесь приведена программа, которая позволяет увидеть эти эффекты в действии.

Листинг 7.9. Эффекты экрана (SCREENFX.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ///////////////////////////////////////
#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <bios.h>
#include <fcntl.h>
#include <memory.h>
#include <malloc.h>
#include <math.h>
#include <string.h>
#include "graph0.h" // включаем нашу графическую библиотеку
// СТРУКТУРА.//////////////////////////////////////////////
typedef struct worm_typ
{
int y;       // текущая Y-координата "червячка"
int color;   // цвет "червячка"
int speed;   // скорость "червячка"
int counter; // счетчик
}, worm, *worm_ptr;
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ //////////////////////////////////
unsigned int far *clock = (unsigned int far *)0x0000046C;
// указатель на внутренний таймер 18.2 "тик"/с
pcx_picture screen_fx; // наш тестовый экран
worm worms[320]; // используется при оплывании экрана
//ФУНКЦИИ ////////////////////////////////////////////////

void Timer(int clicks)
{
// эта функция использует внутренний таймер с частотой 18.2 "тик"/с
// 32-битовое значение этого таймера имеет адрес 0000:046Ch
unsigned int now;
// получим текущее время
now = *clock;
// Ожидаем до истечения указанного периода времени.
// Заметьте, что каждый "тик" имеет длительность примерно в 55 мс
while(abs(*clock - now) < clicks){}
} // конец Timer ////////////////////////////////////////////////////////////
void Fade_Lights (void)
{ // эта функция гасит свет, медленно уменьшая значения цветов
// во всех цветовых регистрах
int index,pal_reg;
RGB_color color,color_1,color_2,color_3;
for (index=0; index<30; index++)
{
for (pal_reg=l; pal_reg<255; pal_reg++)
{
// получить затемняемый цвет
Get_Palette_Register(pal_reg,(RGB_color_ptr)&color) ;
if (color.red > 5) color.red-=3;
else
color.red = 0;
if (color.green > 5) color.green-=3;
else
color.green = 0;
if (color.blue  > 5) color.blue-=3;
else
color.blue = 0;
```

```
// уменьшить интенсивность цвета
Set_Palette_Register(pal_reg,(RGB_color_ptr)&color) ;
} // конец внутреннего цикла
// немного подождем
Timer(2);
} // конец внешнего цикла } // конец Fade_Lights
////////////////////////////////////////////////////////
void Disolve(void)
{
// "растворение" экрана рисованием биллиона черных пикселей
unsigned long index;
for (index=0; index<=300000; index++, Plot_Pixel_Fast(rand()%320, rand()%200, 0));
} // конец Disolve
////////////////////////////////////////////////////////
void Melt(void)
{
// Функция "оплавляет" экран, двигая маленьких "червячков"
// с разной скоростью вниз по экрану. Эти "червячки" меняют
// на своем пути цвет пикселей.
int index,ticks=0;
// инициализация "червячков"
for (index=0; index<160; index++)
{
worms[index].color   = Get_Pixel(index,0);
worms[index].speed   = 3 + rand()%9;
worms[index].y       =0;
worms[index].counter = 0;
// прорисовка "червячка"
Plot Pixel_Fast((index<<1), 0, (char) worms [index].color);
Plot_Pixel_Fast((index<<1), 1, (char) worms [index].color);
Plot_Pixel_Fast((index<<1), 2, (char) worms [index].color) ;
Plot_Pixel_Fast((index<<1) + l,0, (char) worms [index].color) ;
Plot_Pixel_Fast((index<<1) + 1,1, (char) worms [index].color) ;
Plot_Pixel_Fast((index<<1) + 1,2, (char) worms [index].color);
} // конец цикла
// плавим экран
while(++ticks<1800)
{
// работа "червячков"
for (index=0; index<320; index++)
{
// пора подвинуть "червячка"
if (++worms[index].counter == worms[index].speed)
{
// обнуляем счетчик
worms[index].counter = 0;
worms[index].color = Get_Pixel(index,worms[index],y+4);
// достиг "червячок" низа экрана?
if (worms[index].y < 193)
{ Plot_Pixel_Fast ((index<<1), worms [index] .y, 0) ;
Plot Pixel Fast ((index<<3.),worms [index] .y+1,
(char)worms[index].color);
Plot_Pixel_Fast ( (index<<1) ,worms [index] .y+2,
(char)worms[index].color);
Plot Pixel Fast ((index<<1),worms [index] .y+3,
(char)worms[index].color) ;
Plot Pixel Fast ( (index<<1) +1,worms [index] .y, 0) ;
Plot_Pixel_Fast( (index<<1)+1,worms [index] .y+1,
(char)worms[index].color);
Plot Pixel Fast ( (index<<1)+l,worms [index] .y+2,
(char)worms[index].color);
Plot_Pixel_Fast ( (index<<1)+1,worms [index] .y+3,
(char)worms[index].color);
worms[index].y++;
} // конец оператора if
} // конец оператора if
} // конец цикла // ускоряем плавление
```

```
if (!(ticks % 500))
{
for (index=0; index<160; index++) worms[index].speed--;
} // конец оператора if
} // конец оператора while
} // конец Melt
// ОСНОВНАЯ ПРОГРАММА ////////////////////////////////////
void main(void)
(
int index,
done=0,
sel;
// установка видеорежима 320x200x256
Set_Mode(VGA256);
PCX_lnit((pcx_picture_ptr)&screen_fx) ;
PCX_Load("war.pcx", (pcx_picture_ptr)&screen_fx,1);
PCX_Show_Buffer((pcx_picture_ptr) &screen_fx);
PCX_Delete((pcx_picture_ptr)&screen_fx);
_settextposition(22,0);
printf('1 - Fade Lights.\n2 - Disolve.\n3 - Meltdown.");
// какой эффект хочет увидеть игрок? switch(getch())
{
case '1': // гаснущий экран {
Fade_Lights();
} break;
case '2': // растворяющийся экран {
Disolve();
} break;
case '3': // оплывающий экран {
Melt(};
} break;
} //конец оператора switch
// возврат в текстовый режим
Set_Mode(TEXT_MODE) ;
} // конец функции main
```

Пробовали ли вы текстурировать?

Так как основная тема всех наших разговоров — это то, как сделать трехмерную игру типа DOOM с текстурированными поверхностями, то мне следует рассказать вам о несомненно важной вещи, которую мы еще не обсуждали — об этом самом тексту рировании. Все эти темные комнаты и туннели просто обязаны иметь действительно потрясающую текстуру. Здесь есть только одна проблема, а что если вы не художник-профессионал? Как же вы их сделаете?

Вы будете приятно удивлены, узнав что достаточно внимательно посмотреть на всевозможные здания и сооружения, а затем посвятить некоторое время программе Deluxe Paint. Взгляните на рис. 7.8.

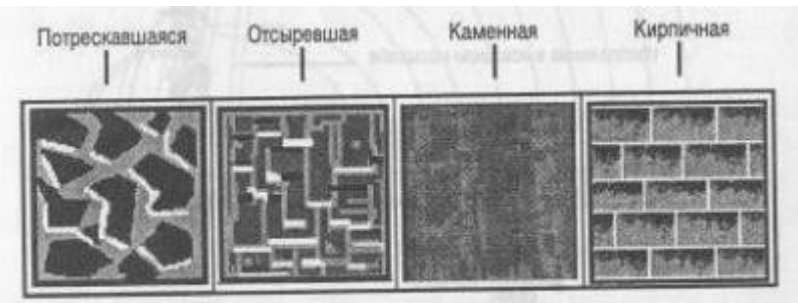


Рис. 7.8. Примеры текстур стен.

Эти несколько грубых текстур я сделал с помощью этой программы за несколько минут. Эти текстуры не так уж плохи, к тому же они показывают, что это можно сделать и не будучи профессионалом. Однако если вы рисуете совсем как кура лапой, то можете воспользоваться тысячей уже готовых текстур с одного из многочисленных дисков CD-ROM.

(Однако, пожалуйста, не размещайте водопроводные краны и ванные на стенах, как это сделано в некоторых играх, названия которых вы сами легко вспомните.)

Масштабирование растровых изображений

Люди обычно говорят: «А вот это мы оставим на сладкое...». В нашем случае мы оставили на сладкое самое сложное, что есть в этой главе. Я пытался свести всю математическую часть к минимуму, зная «любовь» программистов к тригонометрическим функциям и умножению. К сожалению, трехмерные игры содержат довольно сложные алгоритмы и математику. Масштабирование битовых образов - это именно одна из таких областей.

Мы рассматривали в четвертой главе, «Механизмы двухмерной графики», способы масштабирования векторных многоугольников. Для этого мы просто умножали каждую из вершин на коэффициент масштабирования и получали новую координату вершины. Затем новые вершины соединялись векторами и получался новый, уже отмасштабированный образ.

Масштабирование растровых изображении более сложная задача. Проблема состоит в том, что образ состоит из сотен пикселей и для того, чтобы изменить его размер, нам надо увеличить или уменьшить количество пикселей пропорционально коэффициенту масштабирования.

Например, если бы вы захотели увеличить ковбоя с рисунка 7.9 в два раза, то он должен выглядеть так, как это показано на рисунке 7.10.

Рис. 7.9. Исходное растровое изображение.

Рис. 7.10. Изображение с рисунка 7.9, увеличенное в два раза.

Если вы посмотрите внимательно, то увидите, что на рисунке 7.10 ровно в два раза больше пикселей, чем на рисунке 7.9.

Удвоение количества пикселей будет работать, если мы хотим увеличить, образ точнс? в два раза. Однако, если же мы захотим увеличить образ на произвольный коэффициент (как мы будем делать позже при создании нашей трехмерной игры), то нам потребуется более общий алгоритм. Я расскажу вам о восхитительно простом способе масштабирования растрового изображения. Он настолько прост, что просто кажется невозможным!

Давайте подойдем к решению данной проблемы немного с другой стороны. На время забудем об умножении на 0.5 или 2, а подумаем о растягивании и сжатии исходного образа. Другими словами, мы хотим взять исходное количество пикселей и растянуть или сжать их до требуемого количества пикселей. Рисунок 7.11 показывает графическое представление этой идеи.

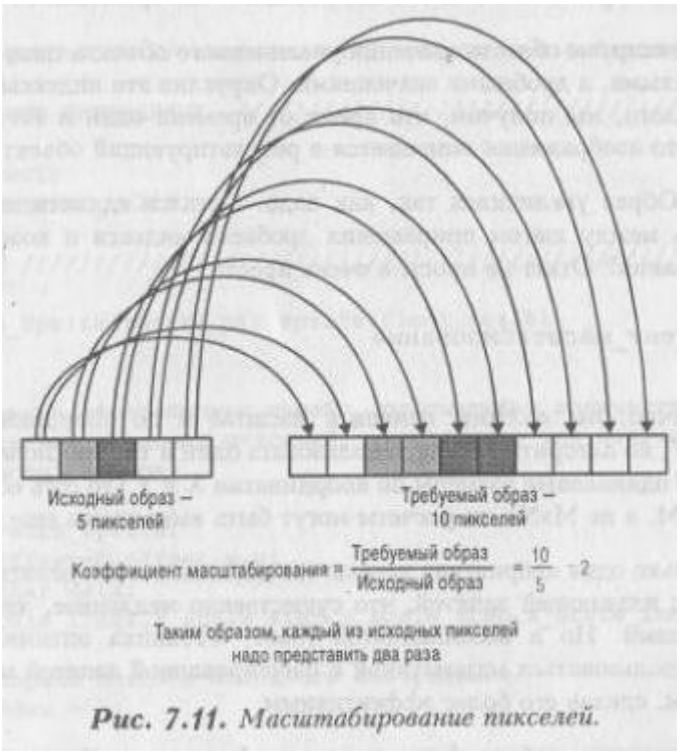


Рис. 7.11. *Масштабирование пикселей.*

Например, если мой исходный образ состоит из строк по 64 пикселя, и я хочу превратить его в образ со строкой по 100 пикселей, то я должен увеличить количество исходных пикселей как 100/64, то есть примерно в 1.5 раза.

Так как мы знаем, что экран представляет собой матрицу целых, то может показаться, что растягивание какого-либо объекта в нецелое количество раз не будет работать. Однако все не так уж и плохо. Мы можем взять часть исходного изображения и затем поместить его на изображение требуемого размера. Затем используя коэффициент обратный коэффициенту масштабирования, определить индекс следующей части масштабируемого изображения и повторить весь процесс сначала. В результате многократного повторения этой процедуры мы и достигнем желаемого результата.

Вернемся к нашему примеру. Мы хотим в результате масштабирования получить 100 пикселей, которые бы выглядели как исходные 64. Как мы уже посчитали, коэффициент масштабирования должен быть 1.5, следовательно, каждый пиксель должен быть скопирован 1.5 раза. Однако мы не можем этого сделать, так как скопировать только половину пикселя при всем желании не удастся. Испробуем наш альтернативный вариант, и сделаем примерно так:

- Проиндексируем область значений исходного изображения от 0 до 63;
- Проиндексируем область значений увеличенного объекта также от 0 до 63, но не целыми, а дробными значениями.

Округлив эти индексы до ближайшего целого, мы получим, что время от времени один и тот же пиксель исходного изображения помещается в результирующий объект дважды.

Вуаля! Образ увеличился так, как надо. Остался единственный вопрос, какова связь между шагом приращения дробного индекса и коэффициентом масштабирования? Ответ не прост, а очень прост:

```
1/коэффициент масштабирования
```

И все! Конечно, мы должны изменять масштаб и по координате X, и по координате Y, но алгоритм можно использовать один и тот же. Более того, если объект имеет одинаковые размеры по координатам X и Y (то есть если он имеет размеры MxM, а не MxN), то расчеты могут быть выполнены еще быстрее.

Есть только одна неприятная деталь: мы должны все это делать, используя математику с плавающей запятой, что существенно медленнее, чем расчеты с целыми числами. Но в восемнадцатой главе, «Техника оптимизации», мы изучим, как пользоваться математикой с фиксированной запятой и перепишем этот алгоритм, сделав его более эффективным.

Чтобы показать работу функции масштабирования объектов я написал демонстрационную программу, которая называется SCALE.C. В ней вы можете выбрать одну из четырех текстур и масштабировать ее, используя клавишу левой угловой скобки (или знака «меньше») для уменьшения и клавишу правой угловой скобки (или знака «больше») — для увеличения объекта. Обратите внимание, что масштабирование действительно замедляется при увеличении объекта, но надо помнить, что это только начало, и нам было важнее всего понять, как все это работает. Листинг 7.10 содержит текст программы. Не забудьте при компоновке программы подключить библиотеку GRAPH0.C.

Листинг 7.10. Программа, масштабирующая текстуры стен (SCALE.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ///////////////////////////////////////
#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <bios.h>
#include <fcntl.h>
#include <memory.h>
#include <malloc.h>
#include <math.h>
#include <string.h>
#include "graph0.h" // включаем нашу графическую библиотеку
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ //////////////////////////////////
sprite object;
pcx_picture text_cells;
// ФУНКЦИИ ////////////////////////////////////////////////
 void Scale_Sprite(sprite_ptr sprite,float scale)
{
// эта функция масштабирует спрайт, рассчитывая количество
// дублирования исходных пикселей, необходимое для получения
// требуемого размера
char far *work_sprite;
int work offset=0, offset,x,y;
unsigned char data;
float y_scale_index,x_scale_step,y_scale_step,x_scale_index;
// берем первый пиксель исходного изображения
y_scale_index = 0;
// рассчитываем дробный шаг
y_scale_step = sprite height/scale;
x_scale_step = sprite_width/scale;
// для простоты даем указателю иа спрайт новое имя
work_sprite = sprite->frames [sprite->curr__frame];
// расчет смещения спрайта в видеобуфере
offset = (sprite->y << 8) + (sprite->y << 6) + sprite->x;
// построчно масштабируем спрайт
for (y=0; y<(int) (scale); y++)
{
// копируем следующую строчку в буфер экрана
x_scale_index=0;
for (x=0; x<(int)scale; x++)
{
// проверка на прозрачность пикселя
// (то есть равен ли он 0), если нет - прорисовываем пиксель
if ((data=work_sprite[work_offset+(int)x_scale_index]))
      double_buffer[offset+x] = data;
x_scale_index+=(x_scale_step);
} // конец внутреннего цикла (по X)
// используя дробный шаг приращения, определяем следующий
// пиксель исходного изображения
y_scale_index+=y_scale_step;
// переходим к следующей строке видеобуфера
//и растрового буфера спрайта
offset      += SCREEN_WIDTH;
work_offset = sprite_width*(int)(y_scale_index);
} // конец внешнего цикла (по У)
} // end Scale_Sprite ////////////////////////////////////////
void Clear_Double_Buffer(void)
// эта функция очищает дублирующий буфер
// несколько грубо, зато работает
_fmemset(double_buffer, 0, SCREEN_WIDTH * SCREEN_HEIGHT +1);
} // конец Clear_Double_Buffer
// ОСНОВНАЯ ПРОГРАММА /////////////////////////////////////
void main(void)
{
int index, done=0 ;
```

```
float scale=64;

// установка видеорежима 320x200x256
Set_Mode(VGA256);
// установка размера спрайта
sprite_width = sprite_height = 64 ;
// инициализация файла PCX, который содержит
// мультиликационные кадры
PCX__Init ((pcx_picture_ptr)&text_cells) ;
// загрузка файла PCX, который содержит мультипликационные кадры
PCX_Load("textures.pcx", (pcx_picture_ptr)&text_cells,1) ;
// PCX_Show_Buffer((pcx_picture_ptr)&text_cells);
Sprite_Init ( (sprite_ptr) &object, 0, 0, 0, 0 ,0, 0);
// выборка четырех интересующих нас текстур
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,
(sprite_ptr)&object,0,0,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,
(sprite_ptr)&object,1,1,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&text cells,
(sprite_ptr)&object,2,2,0) ;
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,
(sprite_ptr)&object,3,3,0) ;
// резервируем память под дублирующий буфер
Init_Double_Buffer() ;
// позиционируем объект в центре экрана
object.curr frame =0;
object.x          = 160-(sprite_width>>1);
object.y          = 100-(sprite height>>1);
// очищаем дублирующий буфер
Clear_Double_Buffer();
// воспроизводим масштабированную текстуру
Scale_Sprite((sprite_ptr)&object,scale) ;
Show_Double_Buffer(double buffer) ;
_settextposition(24,0);
Printf("Q - Quit, < > - Scale, Space - Toggle.");
// главный цикл while(!done)
{
// нажал ли игрок клавишу?
if (kbhit())
{
switch(getch()) {
case '.': // увеличиваем объект
{
if (scale<180) ( scale+=4;
object.x-=2;
object.y-=2;
} // конец оператора if
} break;
case ',' : // уменьшаем объект
{
if (scale>4) {
scale-=4;
object.x+=2;
object.y+=2;
} // конец оператора if
} break;
case ' ': // смена текстуры
{
// текстуры исчерпались?
if (++object.curr_frame==4)
object. curr_frame=0 ;
} break;
case 'q': // до свидания! {
done=1;
) break;
default:break;
} // конец оператора switch
```

```
// очистка дублирующего буфера
Clear_Double_Buffer() ;
// масштабирование спрайта и прорисовка его
//в дублирующем буфере
Scale_Sprite((sprite_ptr)&object,scale);
// вывод дублирующего буфера на экран
Show_Double_Buffer(double_buffer);
_settextpostion(24,0) ;
printf("Q - Quit,"< > - Scale, Space - Toggle.");
}// конец оператора,if
} // конец оператора..while
// удаление файла PCX
PCX_Delete((pcx_picture_ptr)&text_cells);
// возврат в текстовый режим
Set_Mode(TEXT_MODE);
} // конец функции main
```

Повороты объектов

Хотя мы и не собираемся вращать многочисленные растровые изображения, для полноты картины познакомимся с основными принципами этого процесса.

Вращение битовых объектов до сих пор еще не до конца исследованная область. Инженеры и программисты и по сей день пытаются найти наиболее эффективные алгоритмы для этой операции. Проблема заключается в том, что битовый образ - это конгломерат сотни, если не тысячи пикселей и чтобы повернуть образ, нам надо правильно произвести поворот каждой из этих точек, Обычно это связано с довольно сложными математическими преобразованиями, которые выполняются на персональном компьютере не очень-то быстро. Вы можете спросить: «А как же это сделал Крис Роберте в игре Wing Commander?» Очень просто: он заранее получил все возможные повороты всех космических кораблей с помощью программы моделирования, а потом просто занес их в гигантскую таблицу. Единственная операция, которая могла бывыполняться долго ~ масштабирование, тоже была произведена заранее, а результаты также занесены в таблицу. Я всегда действую подобным же образом и вам советую заранее создать все варианты поворотов ваших изображений, используя один из двух пакетов, либо Deluxe Paint фирмы Electronic Arts или, если вам нравится тратить деньги, 3D Studio фирмы AutoDesk. Затем поместите их в таблицу, и пусть ваша программа извлекает изображения из этой таблицы, используя угол поворота, как индекс. Отрицательная сторона этого метода - не очень рациональное использование памяти. Хранение 32 или 64 битных образов для всех возможных объектов отъедает существенный кусок памяти. Однако можно использовать частичные а не полные таблицы. Например, вы можете поместить в таблицу только кадры для одного квадранта, а для других симметричных квадрантов генерировать изображения уже во время игры. Так как поворот вокруг осей X и Y делается очень просто — такой метод используется очень часто.

Чтобы показать пример поворота растрового изображения с помощью таблиц, я написал программу, которая использует нашу функцию масштабирования для файла PCX, содержащего несколько кадров с летящим астероидом. Эта программа, которую я назвал AFIELD.C, передвигает астероид по трехмерному звездному полю, масштабируя его по мере приближения или удаления оигрока. Листинг 7.11 содержит текст этой программы.

Листинг 7.11. Трехмерный астероид (AFIELD.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ////////////////////////////////////////
#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <bios.h>
#include <fcntl.h>
#include <memory.h>
#include <malloc.h>
#include <math.h>
#include <string.h>
#include "graph0.h" // включаем нашу графическую библиотеку
// ОПРЕДЕЛЕНИЯ//////////////////////////////////////////////
#define NUM_STARS 30
// СТРУКТУРЫ ///////////////////////////////////////////////
typedef struct star_typ
{
int x,y;    // позиция звезды
int vel;    // скорость звезды по координате x
int color; // цвет звезды
} star,*star_ptr;
```

```
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ ///////////////////////////////////
star stars[NUM_STARS]; // звездное поле
sprite object;
 pcx_picture ast_cells;
// функции ////////////////////////////////////////
void Star_Field(void) {
static int star_first=1; // Эта функция создает трехмерное звездное поле
int index;
// проверяем, следует ли нам создать звездное поле,
//то есть первый ли раз вызвана функция
if (star_first)
{ // обнуляем флаг первого вызова
star_first=0;
// инициализируем все звезды
for (index=0; index<NUM STARS; index++)
{ // инициализируем для каждой звезды позицию, скорость и цвет
stars[index].x     = rand()%320;
stars[index].y     = rand()%180;
// определяем плоскость звезды
switch(rand()%3){
case 0: // плоскость 1 - самая далекая
{
// установка скорости и цвета
stars[index].vel = 2;
stars[index].color.= 8;
} break;
case 1: // плоскость 2 - среднее расстояние
{
stars[index].vel = 4;
stars[index].color = 7;
) break;
 case 2://плоскость 3 - самая близкая
{
stars[index].vel = 6;
stars[index].color = 15;
} break;
} // конец оператора switch
} // конец цикла
//конец оператора if else
{ // это не первый вызов, поэтому делаем рутинную работу -
// стираем, двигаем, рисуем
for (index=0; index<NUM_STARS; index++)
{
if ((stars[index].x+=stars[index].vel) >=320 ) stars[index].x = 0;
// рисуем
Plot_Pixel_Fast_D(stars[index].x,stars[index].y, stars[index].color);
} // конец цикла
} // конец оператора else
} // конец Star_Field
//////////////////////////////////////////////////////////
void Scale_Sprite(sprite_ptr sprite,float scale)
{
// эта функция масштабирует спрайт, рассчитывая число дублирования
// исходных пикселей, необходимое для получения требуемого размера
char far *work_sprite;
int work_offset=0,offset,x,y;
unsigned char data;
float y_scale_index,x_scale_step,y_scale_step,x_scale_index;
// берем первый пиксель исходного изображения
y_scale_index = 0;
// рассчитываем дробный шаг
y_scale_step = sprite_height/scale;
x_scale_step = sprite_width /scale;
// для простоты даем указателю на спрайт новое имя
work_sprite = sprite->frames[sprite->curr_frame];
// расчет смещения спрайта в видеобуфере
offset = (sprite->y << 8) + (sprite->y << 6) + sprite->x;
```

```
// построчно масштабируем спрайт
for (y=0; y<(int) (scale); y++)
{
// копируем следующую строчку в буфер экрана
x_scale_index=0;
for (x=0; x<(int)scale; x++)
{
// проверка на прозрачность пикселя
//(то есть равен ли он 0), если нет - прорисовываем пиксель
if ((data=work_sprite[work_offset+(int)x_scale_index]))
double_buffer[offset+x] = data;
x_scale_index+=(x_scale_step) ;
} // конец цикла по X
//используя дробный шаг приращения определяем
// следующий пиксель исходного изображения
y_scale_index+=y_scale_step;
// переходим к следующей строке видеобуфера
//и растрового буфера спрайта
offset     += SCREEN_WIDTH;
work_offset = sprite_width*(int)(y_scale_index) ;
} // конец цикла по Y
} // конец Scale Sprite ////////////////////////////////////////////////////////////
void Clear_Double_Buffer(void)
{ // эта функция очищает дублирующий буфер
// несколько грубо, зато работает
_fmemset(double_buffer, 0, SCREEN_WIDTH * SCREEN_HEIGHT + 1);
} // конец Clear_Double_Buffer
// ОСНОВНАЯ ПРОГРАММА //////////////////////////////////
void main(void)
{
int index,
done=0,dx=5,dy=4,ds=4;
float scale=5;
// установка видеорежима 320x200x256
Set_Mode(VGA256) ;
// установка размера спрайта
sprite_width = sprite_height = 47;
// инициализация файла PCX, который содержит
// мультипликационные кадры
PCX_Init((pcx_picture_ptr)&ast_cells) ;
// загрузка файла PCX, который содержит мультипликационные кадры
PCX_Load("asteroid.pcx", (pcx_picture_ptr)&ast_cells,1) ;
// резервируем память под дублирующий буфер
Init_Double_Buffer() ;
Sprite_Init((sprite_ptr)&object,0,0,0,0,0,0) ;
// загрузка кадров вращающегося астероида
PCX_Grap_Bitmap((pcx_picture_ptr)&ast_cells,
(sprite_ptr)&object,0,0,0);
PCX_Grap_Bitmap ((pcx_picture_ptr) &ast_cells,
(sprite_ptr)&object,1,1,0} ;
PCX_Grap_Bitmap((pcx_picture_ptr)&ast_cells,
(sprite_ptr)&object,2,2,0) ;
PCX_Grap_Bitmap((pcx_picture_ptr)&ast_cells,
(sprite_ptr)&object,3,3,0) ;
PCX_Grap_Bitmap ((pcx_picture_ptr) &ast_cells,
(sprite_ptr)&object,4,4,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&ast_cells,
(sprite_ptr)&object,5,5,0) ;
PCX_Grap_Bitmap((pcx_picture_ptr)&ast_cells, (sprite_ptr)&object,6,0,1);
PCX_Grap_Bitmap({pcx_picture_ptr)&ast_cells, (sprite_ptr)&object,1,1,1) ;
// позиционирование объекта в центре экрана
object.curr_frame =0;
object.x          = 160-(sprite width>>1);
object.y          = 100-(sprite_height>>1) ;
// очистка дублирующего буфера
Clear_Double_Buffer();
// вывол масштабированного спрайта
```

```
Scale_Sprite((sprite_ptr)&object,scale) ;
Show_Double_Buffer(double_buffer) ;
// главный цикл
while (!kbhit())
{ // масштабируем астероид
scale+=ds;
// не слишком ли велик или мал астероид?
if (scale>100 |1 scale < 5)
{
ds=-ds;
scale+=ds;
} // конец if
// перемещаем астероид
object.x+=dx;
object.y+=dy;
// коснулся ли астероид края экрана?
if ((object.x + scale) > 310 || object.x < 10)
{
dx=-dx;
object.x+=dx;
} // конец if
if ((object.y + scale) > 190 || object.y < 10) {
dy=-dy;
object.y+=dy;
} // конец if
// поворот астероида на 45 градусов
if (++object.curr_frame==8) object.curr_frame=0; // очистка дублирующего буфера
Clear_Double_Buffer();
// прорисовка звезд
Star_Field() ;
// масштабируем спрайт и выводим его в дублирующий буфер
Scale_Sprite((sprite_ptr)&object,scale) ;
// выводим дублирующий буфер на экран
Show_Double_Buffer (double_buffer);
} // конец оператора while
// удаляем файл PCX
PCX_Delete ((pcx_picture_ptr) &ast_cells);
// возврат в текстовый режим
Set_Mode (TEXT_MODE) ;
} // конец функции main
```

Итог

В этой главе мы затронули много тем, относящихся к серьезной работе с растровой графикой и специальными визуальными эффектами, но, самое главное, мы впервые в этой книге начали разговор о том, как сделать так, чтобы игрок получил от вашей игры *удовольствие.* Я думаю, многие программисты уже забыли, что это такое, променяв его на мегабайты оцифрованного звука и графики.

Поэтому не будем забывать об удовольствии!

ВЫСОКОСКОРОСТНЫЕ ТРЕХМЕРНЫЕ СПРАЙТЫ

На самом деле, новых понятий в этой главе не появится. Скорее, речь пойдет о практическом применении тех технологий и концепций, которые мы обсудили в предыдущих главах. Обычно я никогда не рассматриваю практическую реализацию в книгах подобного плана. Это связано с тем, -что создание программы - скорее, творческий, чем технологический процесс. Однако в данном случае я изменил свое решение, так как хочу, чтобы вы четко разобрались в ряде основополагающих концепций.

За последние несколько лет мир компьютерных игр пережил несколько сенсаций (и появление игры DOOM одна, но не единственная, из них). К этим сенсациям, в том числе следует, отнести и появление на свет игры Wing Commander, написанной Крисом Робертсом (Chris Roberts). Увидев ее впер вые, я был так потрясен, что вскоре она стала моей любимой игрой. В Wing Commander и других играх подобного плана была использована та же технология визуализации спрай тов, что и в играх типа DOOM,

Образы в Wing Commander - это или простые макеты, или создаваемые компьютером объекты, преобразованные в цифровую форму двухмерного спрайта.. При движении по экрану эти спрайты видоизменяются по определенным математическим законам, создавая полное ощущение перспективы. Однако, хотя с формальной точки зрения данные изображения и не являются трехмерными, над ними производятся стандартные процедуры удаления невидимых поверхностей и другие математические расчеты для трехмерных изображений. Другими словами, это проекции трехмерных объектов на плоские многоугольники, которые собственно, и являются спрайтами.

Для изготовления игры типа Wing Commander не требуется сверхсложной математики. Однако подобная игра содержит массу тонкостей, хитрых приемов и замечательных алгоритмов. Посмотрим, сможем ли мы разобраться в некоторых из них, обсудив следующие вопросы:

- Механика трехмерных спрайтов;
- Аксонометрические преобразования;
- Правильный расчет масштаба;
- Видимый объем;
- Новая версия масштабирования;
- Отсечение в трехмерном пространстве спрайтов;
- Построение траекторий;
- Удачный угол зрения на спрайты;
- Трехмерное звездное небо;
- Оцифровка объектов и моделирование;
- Создание съемочной студии;
- Цветовая палитра.

Механика трехмерных спрайтов

Трехмерные спрайты очень похожи на двухмерные. По правде говоря, они и являются плоскими, просто нам кажется, что они объемные. Такое восприятие обуславливается тем, как они движутся по экрану и как при этом изменяется их размер- Двухмерные спрайты могут перемещаться только по осям X и У, трехмерные же должны перемещаться ло всем трем осям — X, У и Z. Движение вдоль оси Z, фактически, осуществляется только в математическом плане за счет изменения координат спрайта одновременно по осям X и Y, а также вычислением его размера, определяемого координатой Z. Именно это и делает плоские картинки похожими на трехмерные.

Конечно, мы опустили некоторые детали. Как может плоское изображение выглядеть трехмерным? Чтобы ответить на этот вопрос, нарисуем объект так; как бы он выглядел под определенным углом зрения в пространстве. В качестве

Рис. 8.1. *Установка для съемки вращения игрушечного космического корабля.*

примера к этой главе, я заснял на видеокамеру вращение игрушечного космического корабля, а затем оцифровал полученные кадры. Для этого я закрепил камеру и объект на определенном расстоянии друг от друга, как показано на рисунке 8.1, и для каждого снимка чуть-чуть поворачивал игрушку вокруг вертикальной оси. Таким образом, я получил 12 кадров, которые изображали корабль под разными углами. Эти кадры приведены на рисунке 8.2. Если мы последовательно отобразим их на экране, то получим иллюзию вращающегося в сотне шагов от нас космического корабля.

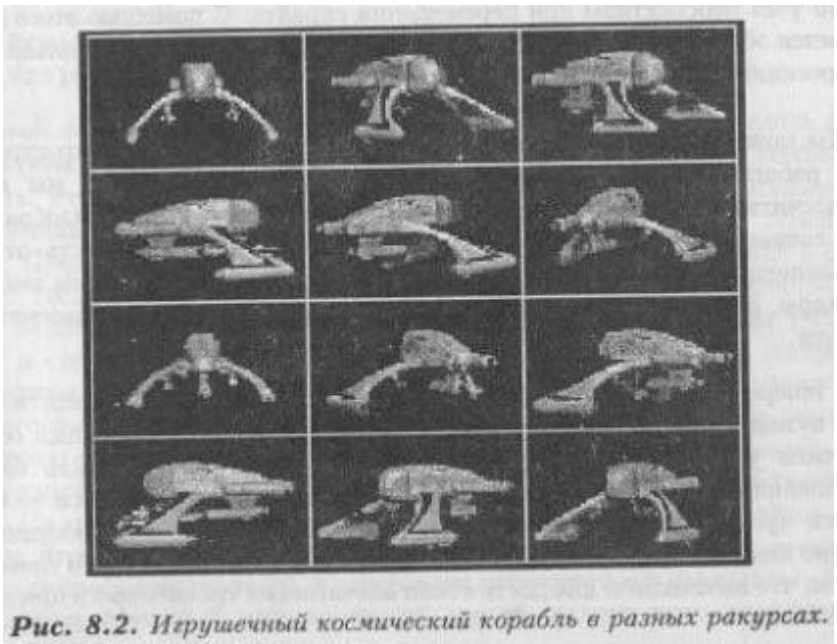


Рис. 8.2. *Игрушечный космический корабль в разных ракурсах.*

Трехмерный вид спрайтов обуславливается двумя различными факторами:

- Во-первых, мы сканируем (или рисуем) объект в различных ракурсах. В конечном счете, лучше всего получить изображение во всех возможных позициях вращения, но это не получится из-за того объема памяти, необходимой для хранения всех изображений. Например, для этой главы я создал спрайты размером 80x48 пикселей. Это значит, что каждое изображение требует 3840 байт. На первый взгляд эта величина не кажется слишком большой, но только до тех пор, пока вы не подсчитаете объем памяти, занимаемый всеми кадрами, выполненными под разными углами зрения. Если мы хотим просто вращать объект относительно оси Y с шагом в 30 градусов и рассматривать его с 4 разных уровней высоты, на это уйдет 12x4x3840=184K. А теперь представим, что нам понадобились изображения большего размера либо несколько подобных изображений (или и то и другое сразу)! Для экономии памяти можно оцифровывать только некоторые из требуемых изображений, а недостающие получать преобразованием имеющихся по мере необходимости. Например, можно нарисовать виды объекта только под углами зрения в пределах от 0 до 180 градусов, а для оставшихся 180 градусов получать изображение из исходного набора данных уже программным путем;
- Второй фактор, с помощью которого достигается иллюзия трехмерности это учет перспективы при перемещении спрайта. С помощью этого достигается эффект передвижения объектов вперед и назад относительно точки проекции и горизонта.

Мы можем считать, что спрайты - это точки в трехмерном пространстве и работать с ними на основе этого предположения. Тогда мы можем рассчитывать коэффициенты их масштабирования и строить изображение в соответствии с расстоянием до них от наблюдателя (то

есть от вас), Наконец, во время перемещения спрайтов по экрану мы должны выбирать кадры, соответствующие углу наблюдения, под которым они рассматриваются.

А теперь поговорим о том, как поддерживать иллюзию объемности. Если игрок, путешествуя по игровому пространству, заходит за трехмерный объект, он должен увидеть его сзади, а отнюдь не с боку, как это было бы при перемещении слева направо. Учет этого обстоятельства основывается на отслеживании траектории объекта и вектора взгляда игрока. Мы обсудим это подробно немножко позже в настоящей главе, а сейчас я просто хотел упомянуть обо всем, что необходимо для достижения впечатления трехмерности при работе со спрайтами.

Аксонометрические преобразования

Мы уже говорили об этом раньше (в шестой главе, «Третье измерением). Для воспроизведения объектов на экране компьютера используются аксонометричес кая проекция и связанные с ней математические преобразования, так как эта проекция выглядит более реалистично, чем параллельная (ортогональная). При построении аксонометрической проекции значения координат X и Y модифицируются с учетом координаты Z (то есть расстояния от наблюдателя до плоскости просмотра, которой в нашем случае является экран), создавая тем самым иллюзию реальной перспективы.

Для правильного аксонометрического преобразования трехмерного спрайта мы должны использовать его центр в качестве его позиции или *локального центра координат.* Это весьма важно! Если мы будем за центр координат принимать верхний левый угол спрайта (как мы это делали раньше), то получим искаженную картину. Всегда в качестве локального центра координат используйте середину объекта. Необходимость этого вызвана тем, что образ должен масштабироваться равномерно от его центра, а не от верхнего левого угла. (Запомнили? Повторите. Отлично!)

Формула 8.1. Аксонометрическая проекция спрайта.

Математический аппарат аксонометрии прост: нам требуется знать только значение координаты Z и расстояние до объекта. Имея эти данные, мы получаем новое положение проекции спрайта следующим образом:

```
x_Projected = view_distance * x_sprite / z_sprite

y_Projected = view_distance * y_sprite / z_sprite
```

Такой метод проекции работает четко: спрайты перемещаются вдоль линий перспективы в окне просмотра. Однако есть одна тонкость, которую необходимо обсудить. Речь идет о том, что я назвал бы *пространственным искажением.* Оно возникает, когда вы смешиваете одну систему виртуальных объектов с другой. Представим, например, что вы помещаете трехмерный спрайт в пространство, созданное трассировкой лучей. По отношению к этому пространству спрайт будет передвигаться и масштабироваться некорректно из-за того, что спрайт и стены создавались на основе двух различных способов построения перспективы. Решить эту проблему можно с помощью всего лишь одного-единственного умножения во время построения проекции. Мы должны умножить координаты спрайта на коэффициент *масштаба внутренней размерности.* Этот поправочный коэффициент отображает одно математическое пространство на другое. По существу, мы должны масштабировать пространство спрайтов таким образом, чтобы оно соответствовало пространству трассированных лучей (или любому другому пространству, в которое мы собираемся его поместить). Конеч но, существует строгий математический способ расчета этого коэффициента (если вы любите точность, напишите мне письмо, и я покажу вам, как это делается). Что же касается меня лично, то я просто играю с расстояниями до спрайтов, пока объекты не начнут выглядеть вполне приемлемо. Итак, мы в общих чертах разобрались с проецированием спрайта на экран. На всякий случай повторю еще раз:

- В качестве начала координат мы используем центр спрайта;
- Затем мы делим координаты спрайта X и Y на координату 2;
- После этого умножаем полученный результат на поправочный коэффициент, учитывающий расстояние до объекта и фактор внутреннего масштаба пространства.

Чем-то напоминает рецепт колдовского зелья, не так ли? Тем не менее, выполнив эти операции и подвигав спрайт по экрану, мы получим иллюзию если не трехмерного, то, по крайней мере, двух с половиной-мерного изображения.

Таким образом, правильно поместив на экран объект, мы должны еще научиться корректно масштабировать его размеры при отдалении (или приближении) спрайта от наблюдателя.

Правильный расчет масштаба

Мы не учли одну вещь: масштаб спрайта. Нам надо увеличивать или уменьшать размеры спрайта, исходя из его положения на оси Z по отношению к наблюдателю.

Другими словами, мы должны отыскать формулу, которая бы позволяла нам получать подходящий размер или масштаб в пикселях для прорисовки спрайта. К счастью, масштаб обратно пропорционален значению Z-координаты спрайта (то есть расстоянию от спрайта до игрока). Следовательно, для вычисления точного масштаба (или, по крайней мере, хорошо смотрящегося масштаба), мы можем использовать Z-координату спрайта с некоторым поправочным коэффициентом.

Опять-таки не надо забывать о пространственных искажениях, возникающих при смешивании одной системы виртуальных объектов с другой. Поправочный коэффициент,

сглаживающий эти искажения, легче всего подобрать экспериментально. Наша масштабирующая подсистема будет вычислять пиксельный размер, в соответствии с которым растровые текстуры должны будут соответственно сжиматься и растягиваться. Сам расчет масштаба на основании Z-координаты спрайта будет производиться по формуле 8.2.

Формула 8.2. Расчет масштабирования.

```
scale=scale_distance/sprite_z,
```

где scale_distance расстояние - визуально дающее хороший результат.

Мы знаем, что создание трехмерной графики не может быть легким делом с ней надо повоевать! И мне это даже нравится! Существует еще одна небольшая проблема - отсечение. И большая часть этой главы посвящена именно этой проблеме.

Видимый объем

Как мы узнали в главах, посвященных трехмерной графике, базирующейся на многоугольниках (глава шестая, «Третье измерение» и седьмая, «Улучшенная битовая графика и специальные эффекты»), объекты должны быть отсечены в пределах видимого объема (или усеченной пирамиды просмотра). Это достигается путем определения ребер каждого многоугольника и отсечения их шестью гранями видимого объема. Возникающая при этом проблема состоит в том, что объем просмотра представляет собой трехмерный трапецоид, состоящий из шести интересующих нас плоскостей, так как это показано на рисунке 8.3.

Мы, конечно, не хотим рассчитывать пересечения спрайтов с произвольными плоскостями. Это было бы уж слишком! Но если мы вначале спроецируем каждый спрайт в аксонометрии и рассчитаем их масштаб, то сможем превратить объем просмотра в прямоугольник. Теперь, так как мы выполняем обратную операцию, мы сможем отсечь спрайты куда как более легким образом, чем при использовании уравнений произвольных плоскостей.

Этот прием основывается на том факте, что объем просмотра уже является аксонометрической проекцией. Если мы отсечем края трехмерного объекта? видимым объемом прежде, чем спроецируем этот объект, мы будем обязаны использовать трапецеидальные формы объема просмотра. Однако, если мы вначале спроецируем объект с учетом перспективы, а *затем* отсечем его прямоугольными границами видимого объема, то.результат, полученный таким образом, будет полностью совпадать с результатом, достигаемым первым способом. Отсечение же в плоскостях, параллельных плоскости просмотра, сводится к простому условному оператору, в котором проверяется, является ли спрайт слишком далеким или слишком близким по отношению к вам.

***Рис. 8.3.** Объем просмотра.*

В случае видеорежима l3h мы можем отсекать все спрайты по размерам экрана или четырехугольника, границы которого определяют точки (0,0) и (319,199). Отсечение в плоскостях, параллельных плоскости просмотра, осуществляется с помощью простого теста на выполнение условия: если спрайт находится внутри наблюдаемого Z-пространства, то визуализируйте объект, а в противном случае игнорируйте его. Отсечение в этих плоскостях выглядит так просто оттого, что в действительности спрайты — это прямоугольные многогранники, расположенные перпендикулярно линии взгляда, или параллельно плоскости просмотра (вы можете думать, как вам больше нравится, но на самом деле это одно и то же).

Я полагаю, что, поэкспериментировав с отсечением дальней плоскостью, вы сделаете так, чтобы объект размером меньше, чем в один пиксель по любому измерению (после масштабирования) вообще не рисовался. Выбирая ближнюю плоскость отсечения, старайтесь сделать так, чтобы приближающийся объект не выглядел как под микроскопом с многократно увеличенными пикселями.

Новая версия масштабирования

Прежде чем попытаться реализовать алгоритм отсечения, мы должны разобраться с основами операций масштабирования. Поскольку масштабирование и отсечение идут рука об руку, мы должны сначала написать программу изменения масштаба, а затем адаптировать ее для процесса отсечения.

Механизм масштабирования мы показали в седьмой главе, «Улучшенная битовая графика и спецэффекты» (Листинг 7.9). Для масштабирования текстуры стен в этом алгоритме был

использован не совсем обычный подход. Каждая стена рассматривалась не как обычная двухмерная матрица, имеющая высоту и ширину, а разбивалась на отдельные одномерные столбцы. В результате программе нужно было масштабировать только столбцы шириной в один пиксель. (Кстати, простой текстурированный пиксель называется текстелем (textel).)

Для реализации двухмерного масштабирования мы слегка модифицируем код, представленный в Листинге 7.9. В новом варианте будут использованы только целые числа, а индексы масштабирования будут предварительно подсчитаны и занесены в таблицу соответствий. Текст новой программы масштабирования приведен в Листинге 8.1.

Листинг 8.1. Новая функция масштабирования спрайтов (без отсечения).

```
void Scale_Sprite(sprite_ptr sprite,int scale)
//Эта функция масштабирует спрайт (без отсечения). Масштабирование
// производится с использованием заранее рассчитанной таблицы,
// которая определяет, как будет изменяться каждый вертикальный
// столбец. Затем другая таблица используется для учета
// масштабирования этих столбцов по оси X
char far *work_sprite; // текстура спрайта
int *row_y;             // указатель на масштабированные
         // по оси Y данные (заметьте, что это
            // ближний указатель)
int far *row_x;         // указатель на масштабированные
   // по оси X данные (заметьте, что это
            // дальний указатель)
unsigned char pixel;    // текущий текстель
int    x,                   // рабочие переменные
y,
column, work_offset, video_offset, video_start;
// если объект слишком мал, то и рисовать его не стоит
if (scale<1) return;
// рассчитываем необходимые для масштабирования данные
row_y = scale_table_y [scale];
row_x = scale_table_x[scale];
// выбираем соответствующий кадр спрайта
work_sprite = sprite->frames[sprite->curr_frame];
// рассчитываем начальное смещение
video_start = (sprite->y << 8) + (sprite->y << 6} + sprite->x;
// изображение рисуется слева направо и сверху вниз
for (x=0; x<scale; x++)
{
// пересчитываем адрес следующего столбца
video_offset = video_start + x;
// определяем, какой столбец должен быть отображен,
//исходя из индекса масштабирования по оси X
column = row_x[x];
// наконец рисуем столбец обычным образом
for (y=0; y_scale; y++)
{
// проверка на "прозрачность"
pixel = work_sprite[work_offset+column];
if (pixel)
double_buffer[video_offset] = pixel;
// индекс следующей строки экрана и смещение в области
// хранения текстуры
video_offset += SCREEN_WIDTH;
work_offset = row_y[y] ;
} // конец цикла по У
} // конец цикла по X
} // конец Scale_Sprite
```

Как видите, это простая и короткая функция. Это достигается благодаря использованию двух таблиц масштабирования. В них расположены индексы масштабирования: в одной — для масштабирования по координате X, а в другой - по У. Две таблицы нужны на тот случай, если ширина и длина спрайта окажутся не одинаковыми. Таким образом, если спрайт всегда имеет размеры MxM, то алгоритм масштабирования может быть еще более упрощен.

Отметим, что таблицы соответствия находятся в разных сегментах памяти: ближнем (NEAR) и дальнем (FAR). Это сделано для скорости. Таблица соответствия во внутреннем цикле масштабирования (по оси У) должна быть в ближнем сегменте данных для ускорения доступа. Таблица соответствия во внешнем цикле (масштабирование по оси X) может располагаться в дальнем сегменте, так как доступ к ней обычно осуществляется только несколько десятков раз.

В общем, было бы, конечно, лучше поместить обе таблицы в ближнем сегменте данных. Однако в результате этого можно очень быстро исчерпать его, что осложнит размещение глобальных переменных. Основное правило при работе с таблицами соответствий на языке Си состоит в следующем: пытайтесь расположить наиболее часто используемые таблицы в ближнем сегменте, а менее употребимые — в дальнем. Это правило не касается того случая, когда есть возможность уместить и таблицу, и глобальные переменные в 64K ближнего сегмента данных.

Теперь посмотрим на нашу новую процедуру масштабирования в действии. Я написал программу, которая загружает ряд заранее отсканированных изображений размером 80x48 пикселей. Эти изображения были сделаны с помощью макета космического корабля, который фотографировался под разными углами в моей импровизированной студии (о ней я уже рассказывал в начале главы). Эта программа (VYREN.C) показывает вращающийся на некотором расстоянии от наблюдателя (то есть от вас) космический корабль и позволяет передвигать его по оси Z с помощью клавиш > (правая угловая скобка или «меньше») и < (левая угловая скобка или «больше»), (На самом деле, в нижнем регистре это будут клавиши с символами запятой и точки соответственно.) Вы увидите, что новый метод с использованием таблиц существенно быстрее прежнего.

Текст этой программы приведен в Листинге 8.2. Потратьте некоторое время и постарайтесь понять, как работает эта программа. Как обычно, программа должна быть скомпонована с нашей растущей графической библиотекой, GRAPHICS.C, и скомпилирована с использованием модели памяти MEDIUM. Для завершения работы программы, следует нажать клавишу Q.

Листинг 8.2. Демонстрационная программа новой функции масштабирования (VYREN.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ///////////////////////////////////////
#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <bios.h>
#include <fcntl.h>
#include <memory.h>
#include <malloc.h>
#include <math.h>
#include <string.h>
#include <graph.h>
#include "graphics.h" // включаем нашу графическую библиотеку
// ПРОТОТИПЫ ///////////////////////////////////////////////////
void Create_Scale_Data_X(int scale, int far *row);
void Create_Scale_Data_Y(int scale, int * row);
void Build_Scale_Table(void);
void Scale_Sprite(sprite_ptr sprite,int scale);
void Clear_Double_Buffer(void) ;
// ОПРЕДЕЛЕНИЯ /////////////////////////////////////////////
#define MAX_SCALE      200      // число звезд на звездном небе
#define SPRITE_X_SIZE  80       // максимальные размеры
#define SPRITE_Y_SIZE  48       // растрового изображения
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ ///////////////////////////////////
sprite object;                      // обобщенный спрайт, который
// содержит кадры с космическим
// кораблем
pcx_picture text_cells;// PCX-файл с изображениями
int *scale_table_y[MAX_SCALE+l];     // таблица с предварительно
// рассчитанными коэффициентами
// масштабирования
int far *scale_table_x[MAX_SCALE+l]; // таблица с предварительно
          // рассчитанными коэффициентами
          // масштабирования
// ФУНКЦИИ ////////////////////////////////////////////////
void Create_Scale_Data_X(int scale, int far *row)
{
// эта функция масштабирует полосу текстуры для всех возможных
// размеров и создает огромную таблицу соответствий
int x;
float x_scale_index=0, x_scale_step;
// рассчитываем шаг масштабирования или число исходных пикселей
// для отображения на результирующее изображение за цикл
x_scale_step = (float)(sprite_width)/(float)scale;
x_scale_index+=x_scale_step;
```

```
for (x=0; x<scale; x++)
{
// помещаем данные в массив для последующего использования
row[x] = (int)(x_scale index+,5);
if  (row[x] > (SPRITE_X_SIZE-1)) row[x] = (SPRITE_X_SIZE-1);
// рассчитываем следующий индекс
x_scale index+=x_scale_step;
} // конец цикла
} // конец Create_Scale_Data_X
//////////////////////////////////////////////////////
void Create_Scale_Data Y(int scale, int *row)
{
// эта функция масштабирует полосу текстуры для всех возможных
// размеров и создает огромную таблицу соответствий
int y;
float y_scale_index=0, y scale_step;
// рассчитываем шаг масштабирования или число исходных пикселей
// для отображения на результирующее изображение за цикл
y_scale_step = (float)(sprite_height)/(float)scale;
y_scale index+=y_scale_step;
for (y=0; y<scale; y++)
{
// помещаем данные в-массив для последующего использования
row[y] = ((int)(y_scale_index+.5)) * SPRITE_X_SIZE;
if (row[y] > (SPRITE_Y_SIZE-1)*SPRITE_X_SIZE) row[y] = (SPRITE_Y_SIZE-1)*SPRITE_X_SIZE;
// рассчитываем следующий индекс
y_scale_index+==y_scale_step;
} // конец цикла
} // конец Create_Scale_Data_Y ////////////////////////////////////////
void Build_Scale_Table (void)
{ // эта функция строит таблицу масштабирования путем расчета
// коэффициентов  масштабирования для всех возможных размеров
// от 1 до 200 пикселей
int scale;
// резервируем память
for (scale=l; scale<=MAX_SCALE; scale++)
{
scale_table_y[scale] = (int *)malloc(scale*sizeof(int)+1);
scale_table_x[scale] = (int far *)_fmalloc(scale*sizeof(int)+l);
} // конец цикла
// создаем таблицу масштабирования для осей X и Y
for (scale=l; scale<=MAX_SCALE; scale++) {
// рассчитываем коэффициент для данного масштаба
Create_Scale_Data_Y(scale, (int *)scale_table_y[scale]);
Create_Scale_Data_X(scale, (int far *)scale_table_x[scale]) ;
}// конец цикла
}// конец Build_Scale_Table ////////////////////////////////////////////////////////
void Scale_Sprite(sprite_ptr sprite,int scale)
{
// эта функция масштабирует спрайт (без отсечения). Масштабирование производится с
//использованием заранее рассчитанной таблицы, которая определяет, как будет
изменяться //каждый вертикальный столбец. Затем другая таблица используется для учета
//масштабирования этих столбцов по оси X
char far *work_sprite;  // текстура спрайта
int *row_y;             // указатель на масштабированные
// по оси Y данные (заметьте, что
// это ближний указатель)
int far *row_x;         // указатель на масштабированные
// по оси X данные (заметьте, что
// это дальний указатель)
unsigned char pixel;    // текущий текстель
lnt x,                  // рабочие переменные
У, column,
work_offset,
video_offset,
video_start;
// если объект слишком мал, то и рисовать его не стоит
```

```
if (scale<1) return;
// рассчитываем необходимые для масштабирования данные
row_y = scale_table_y[scale];
row_x = scale table_x[scale];
// выбираем соответствующий кадр спрайта
work_sprite = sprite->frames[sprite->curr_frame];
// рассчитываем начальное смещение
video_start = (sprite->y << 8) + (sprite->y << 6) + sprite->x;
// изображение рисуется слева направо и сверху вниз
for (x=0; x<scale; x++)
{
// пересчитываем адрес следующего столбца
video_offset = video_start + x;
// определяем, какой столбец должен быть отображен,
// исходя из индекса масштабирования по оси X
column = row_x[x];
// Наконец рисуем столбец обычным образом
for (y=0; y<scale; y++)
{
// проверка на "прозрачность"
pixel = work_sprite[work_offset+column] ;
if (pixel)
double buffer[video_offset] = pixel;
// индекс следующей строки экрана и смещение
//в области хранения текстуры
video_offset += SCREEN_WIDTH;
work_offset  =  row_y[y];
} // конец цикла по Y
} // конец цикла ро X
} // конец Scale_Sprite
////////////////////////////////////////
void Clear Double_Buffer(void) {
// угадали что это?
_fmemset(double_buffer, 0, SCREEN__WIDTH * SCREEN_HEIGHT + 1);
} // конец Clear_Double_Buffer
// ОСНОВНАЯ ПРОГРАММА ////////////////////////////////////
void main(void)
{
// Загружаем 12 предварительно отсканированных кадров спрайта
// и последовательно меняем их до тех пор пока игрок не изменит
// координату Z объекта, нажав клавишу "," или "."
int done=0,                     // флаг завершения
count=0,                   // счетчик времени изменения кадра
scale=64;                  // текущий масштаб спрайта
float scale_distance = 24000, // произвольная константа
// для согласования
// плоской текстуры и трассированного
// пространства
view_distance = 256, // дистанция до объекта
x=0,                       // позиция корабля в трехмерном
           // пространстве
y=0,
z=1024;
// установка видеорежима 320x200x256
_setvideomode(_MRES256COLOR) ;
sprite_width = 80;
sprite_height =48;

// создание таблицы для подсистемы масштабирования
Build_Scale_Table ();
// инициализация файла PCX, содержащего кадры
PCX_Init((pcx_picture_ptr)&text cells) ;
// загрузка файла PCX, содержащего кадры
PCX_Load("vyrentxt.pcx", (pcx_picture_ptr)&text_cells,1);
// резервируем память под дублирующий буфер
Init_Double_Buffer ();
Sprite_Init((sprite_ptr)&object,0,0,0,0,0,0) ;
```

```
// загружаем 12 кадров с космическим кораблем
PCX_Grap_Bitmap ( (pcx_picture_ptr) &text_cells,
(sprite_ptr)&object,0,0,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,
(sprite_ptr)&object,1,1,0) ;
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,
(sprite_ptr)&object,2,2,0) ;
PCX_Grap_Bitmap ((pcx_picture_ptr)&text_cells,
(sprite_ptr)&object,3,0,1);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,4,1,1) ;
PCX_Grap_Bitmap ((pcx_picture_ptr)&text_cells,
(sprite_ptr)&object,5,2,1);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,
(sprite_ptr)&object,6,0,2);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,
(sprite_ptr)&object,7,1,2) ;
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,
(sprite_ptr)&object,8,2,2) ;
PCX_Grap_Bitinap( (pcx_picture_ptr)&text_cells,
(sprite_ptr)&object,9,0,3);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,
(sprite_ptr)&object,10,1,3);
PCX__Grap_Bitmap( (pcx_picture_ptr) &text_cells,
(sprite_ptr)&object,11,2,3) ;
// начальная позиция корабля
object.curr_frame =0;
object.x          = 0;
object, y          =0;
Clear_Double_Buffer();
// ожидаем нажатия клавиш и рисуем корабль
while(!done)
{
// нажал ли игрок клавишу?
if (kbhit())
{
switch(getch()) {
case '.': // отдаляем корабль
{
z+=16;
} break;
case ',':// приближаем корабль
{
z-=l6;
//не позволяем кораблю подойти слишком близко
if(Z<256)
z=256;
} break;
case 'q': // выход из программы
{
done=1;
} break;
default:break;
} // конец оператора switch
} // конец оператора if
//рассчитываем размер растрового изображения
scale = (int)( scale_distance/z );
// исходя из размера растрового изображения,
// рассчитываем проекции координат X и Y
object.x= (int)((float)x*view_distance / (float)z) + 160 - (scale>>1);
object.y = 100  - (((int)((float y*view_distanc=e / (float)z) + (scale>>1)));
// увеличиваем счетчик кадров
if (++count==2)
{
count=0;
if (++object.curr_frame==12) object.curr_frame=0 ;
} // конец оператора if
// очищаем дублирующий буфер
```

```
Clear_Double_Buffer();
// масштабируем спрайт
Scale_Sprite((sprite_ptr)&object,scale);
Show_Double_Buffer(double_buffer);
// выводим информацию на экран
_settextposition(24,0) ;
printf("z Coordinate is %f",z);
} // конец оператора while
// Удаляем файл PCX
PCX_Delete((pcx_picture_ptr) &text_cells);
// восстанавливаем текстовый режим
_setvideomode(_DEFAULTMODE);
} // конец функции main
```

После выполнения программы из Листинга 8.2, вы, возможно, будете удивлены возможностями оцифровки изображений макетов и приведенным вариантом трехмерной мультипликации. Кто знает, может быть вы создадите что-нибудь именно по типу игры Wing Commander, а вовсе не очередную вариацию DOOM? Однако пора переходить к алгоритму отсечения.

Отсечение спрайтов в трехмерном пространстве

После построения аксонометрической проекции спрайта отсечение выполняется довольно легко. Алгоритм просто тестирует, не выходит ли проекция отмасштабированного спрайта за границы экрана, и, кроме того, проверяет, находится ли Z-координата спрайта внутри наблюдаемой области пространства. Следовательно, проблема отсечения сводится к проверке расположения прямоугольника относительно границ экрана. Решение этой проблемы мы уже рассмотрели раньше (в четвертой главе, «Механизмы двухмерной графики»).

Как вы помните, существует два подхода к этой проблеме: можно использовать алгоритм пространства образов и алгоритм пространства объектов. Первый путь намного проще. Перед обновлением каждого пикселя спрайта, мы проверяем, находится ли он внутри границ экрана (или окна просмотра), и если это так, то замещаем пиксель. Недостаток этого метода — низкая производи тельность (хотя иногда все же приходится прибегать к алгоритму пространства образов из-за чересчур сложной геометрической формы визуализируемого объекта). В общем же случае алгоритм пространства образов работает медленнее алгоритма пространства объектов.

При использовании объектно-пространственного алгоритма мы должны каким-то образом до прорисовки определить, какая часть спрайта будет нарисована и на основании этого вновь вычислить его проекцию. По существу, мы должны отсечь границами экрана прямоугольник, который получается в результате масштабирования спрайта. Это кажется несложным. Мы разбирали текст такой программы в предыдущей главе (Листинг 7.3), но я повторю этот алгоритм еще один раз! Такой уж я.

Алгоритм 8.1 предполагает, что:

- Экран ограничивается точками (0,0) и (scrfeen_x, screen_y);
- Верхняя левая точка спрайта (sprifce_x, sprite_y);
- Спрайт имеет размеры width и height.

Предположим, отсечение пространства по оси Z уже было сделано и внешне образ выглядит вполне правдиво. Пусть также были сосчитаны масштаб объекта и координаты проекций по осям X и Y.

Алгоритм 8.1. Масштабирование спрайта.

```
// Проверка полной невидимости спрайта
// (то есть лежит ли он полностью за пределами экрана)
if ((sprite_x > SCRESN_X} or (sprite_y > SCREEN_Y)
or (sprite_x + Width < 0) or (sprite_y + Heigth < 0))
{
// ничего не делаем
return;
} // конец if
else
{
// Спрайт виден частично, следовательно,
// необходимо рассчитать рисуемую область
// Задаем область спрайта перед отсечением
start_x = sprite_x;
start_y = sprite_y;
end_x = sprite_x+ Width - 1;
end_y = sprite_y + Height - 1;
// Отсекаем область слева и сверху
if (sprite_x < 0) start x = 0;
if (sprite_y < 0) start_y = 0;
// Отсекаем область справа и снизу
if (sprite_x + Width > SCRESN_X) end_x = SCREEN_X;
if (sprite_y + Height > SCREEN_Y) end_y = SCREEN_Y;
// Теперь новый спрайт будет иметь координаты верхнего
// левого угла (start_x, start_y), а нижнего правого
// (end x, end y). Эти координаты будут использоваться
// при прорисовке спрайта
return;
} // конец else
```

Как и в любом алгоритме, массу деталей я опустил. Однако идею, я думаю, вы поняли.

Теперь перейдем к самой сложной части отсечения спрайтов: когда мы их масштабируем, то должны взять некоторое число пикселей и «перенести» их на место назначения (экран) столько раз, сколько это необходимо для получения необходимого размера изображения. Однако после масштабирования спрайта его экранный размер может измениться. Поэтому при изображении спрайта после масштабирования мы должны особенно тщательно следить за отсечением. Само по себе масштабирование будет производиться точно так же как и раньше, но рисоваться будет только видимая часть изображения. Другими словами, рассчитав коэффициент масштабирования (или, построив таблицу соответствия), мы будем выводить на экран только часть спрайта, поскольку если изображение будет рисоваться целиком, то оно может выйти за пределы экрана и оказаться не отсеченным.

Построение траекторий

Пусть мы имеем отмасштабированные и отсеченные нужным образом спрайты, и теперь перед нами стоит проблема их передвижения в мире игры. Это означает, что мы должны вычислить траектории движения наших спрайтов. Зная траектории, мы также сможем определить угол между направлением движения объекта и направлением взгляда наблюдателя и выбрать подходящие кадры для вывода на экран. Это будет создавать иллюзию объемности при перемещении спрайтов по экрану.

Траектория — это на самом деле вектор. Для построения вектора необходимо иметь начальную и конечную точки. Так как в нашем случае траектория — это вектор скорости, мы должны иметь исходную точку в начале координат, как показано на рисунке 8.4.

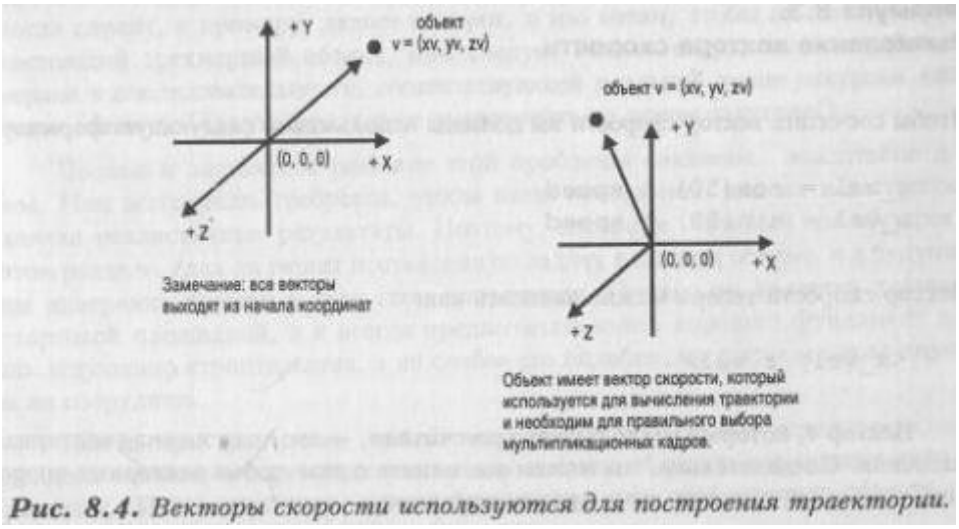


Рис. 8.4. *Векторы скорости используются для построения траектории.*

Конечная точка содержит информацию двух типов;

- Во-первых, направление траектории или скорости;
- Во-вторых, величину линейной или угловой скорости.

Построить вектор скорости просто. Пусть космический корабль движется в направлении 50 градусов в плоскости X-2 (см. рис. 8.5).

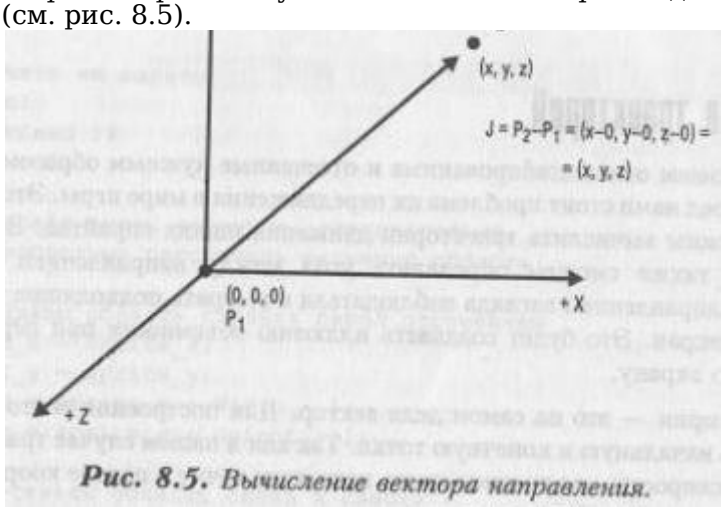


Рис. 8.5. *Вычисление вектора направления.*

Формула 8.3. Вычисление вектора скорости.

Чтобы сосчитать вектор скорости вы должны использовать следующую формулу:

```
x_vel = cos(50) * speed
z_vel = sin (50) * speed
```

Вектор скорости теперь можно записать как: V == <x_vel, z_vel>

Вектор V, который мы только что рассчитали, - это полноценная векторная, величина. Следовательно, мы можем выполнять с ним любые векторные операции, например, сложить его с другим вектором.

Замечание

Отметим, что вектор V лежит в плоскости X-Z, или в плоскости основания по отношению к экрану. Вы, возможно, и не заметили, что в Wing Commander корабли перемещаются, в основном, в этой плоскости, а не в полном X-Y-Z объеме.

Компоненты вектора скорости при перемещении корабля или других объектов используются для преобразования их координат, выступая в роли коэффициентов этого преобразования. Естественно, мы сможем изменять только координаты X и Z, так как только эти компоненты входят в уравнение нашего вектора скорости. Следовательно, в общем случае, чтобы осуществлять преобразование по всем направлениям, мы должны создать трехкомпонентный вектор скорости.

Если вы захотите, вектор скорости может генерироваться случайно на основании данных, введенных игроком или полученных из файла. Но в любом случае вам необходимо знать направление движения объекта (или его главный вид), чтобы точно определить соответствующий кадр для вывода на экран. Ведь объекты представляются с помощью спрайтов, которые на самом деле плоские!

Удачный угол зрения на спрайты

Эта тема наиболее сложная из всех, которые мы до сих пор рассматривали. При перемещении трехмерных спрайтов по экрану, с точки зрения наблюдателя все должно выглядеть так, словно объекты и в самом деле имеют три измерения. Наблюдатель в нашем случае — это игрок, и его позиция фиксирована. Обычно она имеет координаты (0,0,0) или слегка сдвинута по оси Z. В любом случае, когда спрайт, к примеру, делает поворот, и мы хотим, чтобы он выглядел как настоящий трехмерный объект, нам следует менять кадры этого спрайта на экране в последовательности, соответствующей реальной смене ракурсов «живого» объекта. (Такую фразу и не выговоришь на одном дыхании!)

Полное и элегантное решение этой проблемы слишком... масштабно для нас. Нам всего лишь требуется, чтобы наши программы, модели и алгоритмы. давали реалистичные результаты. Поэтому, алгоритм, который мы обсудим в этом разделе, едва ли решит поставленную задачу в полном объеме, и в будущем, вы наверняка внесете в него свои дополнения. Однако он является хорошей стартовой площадкой, а я всегда предпочитаю иметь хороший фундамент для последующего строительства, а не слабое его подобие, на котором после ничего и не соорудишь.

Как и при написании любой программы для любого компьютера, мы должны вначале четко определить задачу, а затем обдумать возможные пути ее решения. • Итак, проблема: как выбрать кадр для изображения объекта на основании угла между лучом зрения игрока и направлением «взгляда» самого объекта.

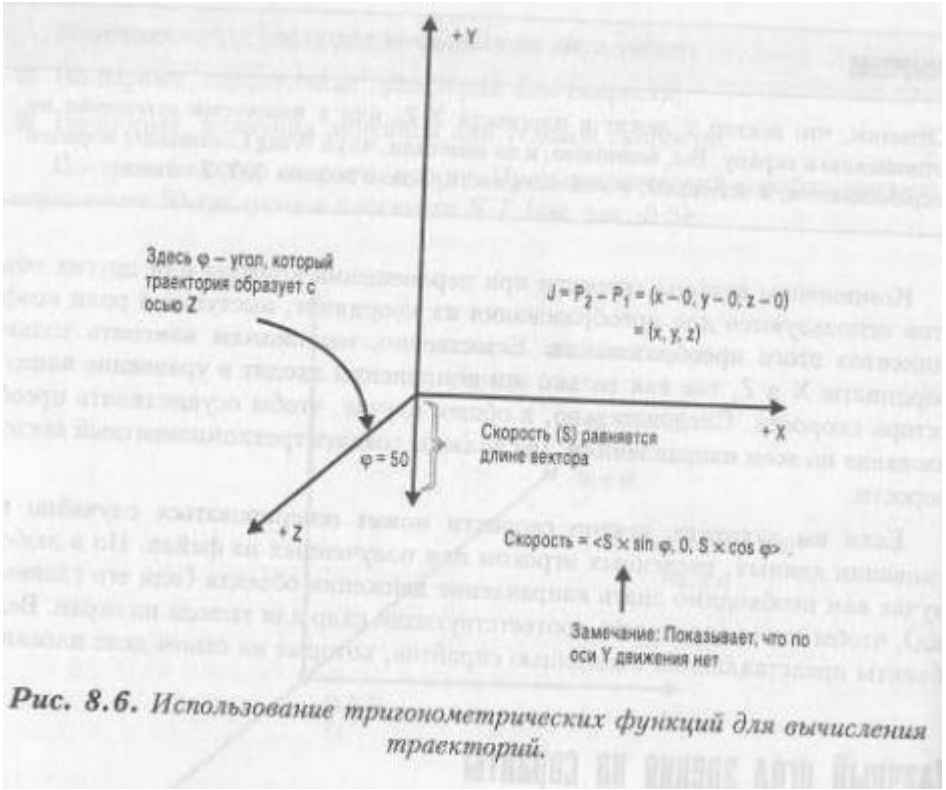


Рис. 8.6. *Использование тригонометрических функций для вычисления траекторий.*

Замечание

В общем случае объект может передвигаться и не в том направлении, куда обращена его лицевая сторона, Мы не будем сейчас заострять внимание на этом варианте движения, чтобы чрезмерно не усложнять нашу задачу. Пока будем считать, что объект всегда движется в ту сторону, куда он обращен лицом. Это предположение вполне обосновано, так как в программе, которую мы впоследствии напишем, будут участвовать космические корабли с кормовыми дюзами. Точно так же и в играх типа Wolfenstein или DOOM игровые персонажи обычно движутся в том направлении, куда они смотрят (или в обратном, если они пятятся).

Попытаемся вначале проанализировать проблему. Нам необходимо определить вид объекта, который зависит от направления взгляда игрока и траектории объекта или направления его движения. Как мы уже говорили, луч зрения игрока можно зафиксировать и считать, что он всегда перпендикулярен экрану. Тогда нам нужно будет побеспокоиться только о векторе траектории объекта, выводимого на экран. На рисунке 8.7 изображена взаимосвязь между вектором направления взгляда игрока и некоторой траекторией передвижения объекта.

Рис. 8.7. *Угол между вектором направления взгляда игрока и траекторией движения объекта игры.*

Теперь мы должны сделать вот что: возьмем игрушечную машинку или что-нибудь подобное и будем передвигать ее перед собой (шум мотора имитировать при этом не обязательно, можно все делать тихо). Проделав это, вы быстро придете к выводу, что рисуемое на экране изображение космического корабля, движущегося прямолинейно, практически одинаково для всех параллельных траекторий независимо от местоположения объекта. Конечно, это справедливо только частично, зато мы получили хорошую отправную точку для нашего первого алгоритма выбора правильного кадра.

Что же мы должны сделать:

- Вычислить угол между траекторией движения объекта и лучом зрения игрока (который всегда направлен прямо в экран);
- Разделить полученный угол на квадранты. Затем на основании полученного индекса выбрать наиболее подходящее изображение среди предварительно подготовленных оцифровкой фотографий модели или нарисованных в графическом редакторе. (Более подробно это обсуждается в разделе «Оцифровка объектов и моделирование».)
- Вывести на экран подходящий кадр, используя аксонометрическую проекцию и масштабируя объект до необходимого размера.

В результате на экране получается реалистичная картина.

Каким же образом находится угол между траекторией объекта и лучом зрения наблюдателя? Ответ может быть получен с помощью *скалярного произведения векторов.* Мы знаем, что угол между двумя векторами можно найти с помощью скалярного произведения векторов, как это показано на рисунке 8.8.

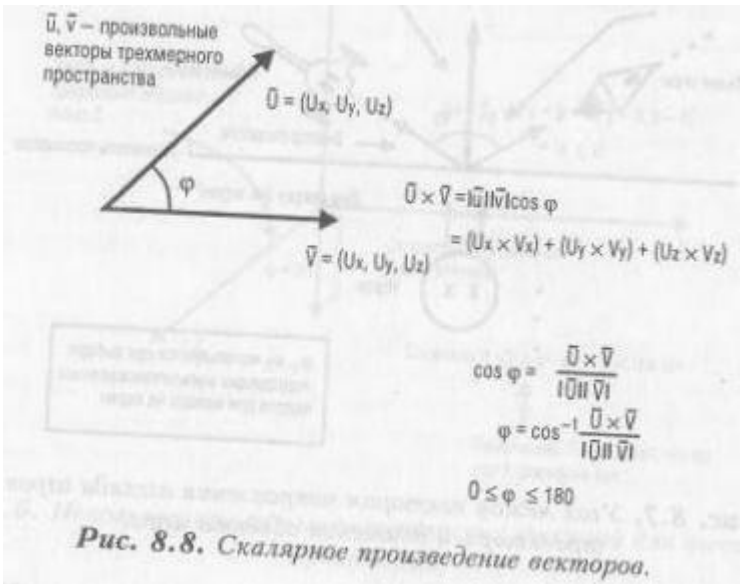


Рис. 8.8. *Скалярное произведение векторов.*

Формула 8.4. Вычисление угла между наблюдателем и объектом.

Если мы зададим вектор направления взгляда, как V, а вектор скорости, как O, тогда угол между ними можно будет найти по следующей формуле:

Пусть V = (vx,vy,vz) и O = (ox,oy,oz), тогда

$$\cos \frac{(vx \times ox) + (vy \times oy) + (vz \times oz)}{\sqrt{vx^2 + vy^2 + vz^2} \times \sqrt{ox^2 + oy^2 + oz^2}}$$

Если бы мы хотели сформулировать это действие словами, то могли бы сказать так: «Угол между V и O равен арккосинусу скалярного произведения этих векторов, разделенного на произведение длин векторов».

Угол между V и O, рассчитанный по этой формуле, имеет одну особенность: он всегда внутренний, то есть больше 0, но меньше 180 градусов. Следовательно, один и тот же результат, полученный по этой формуле, может соответствовать двум разным углам. Это происходит потому, что скалярное произведение не дает информации о направлении вектора (или о направлении, в котором вы отсчитываете положительный угол). Другими словами, эта формула всегда выдает наименьший из углов между двумя векторами. Если вы будете помнить об этом, то такое поведение данной формулы не будет большой проблемой. (Это напоминает бутерброд, который всегда падает маслом вниз. Если вы не знаете об этом, то такой результат может свести вас с ума. А кто предупрежден, тот вооружен.)

Рисунок 8.9 иллюстрирует указанную проблему графически. На этом рисунке показан вектор направления взгляда, три возможных положения вектора траектории и полученный в результате расчетов по формуле 8.4 угол.

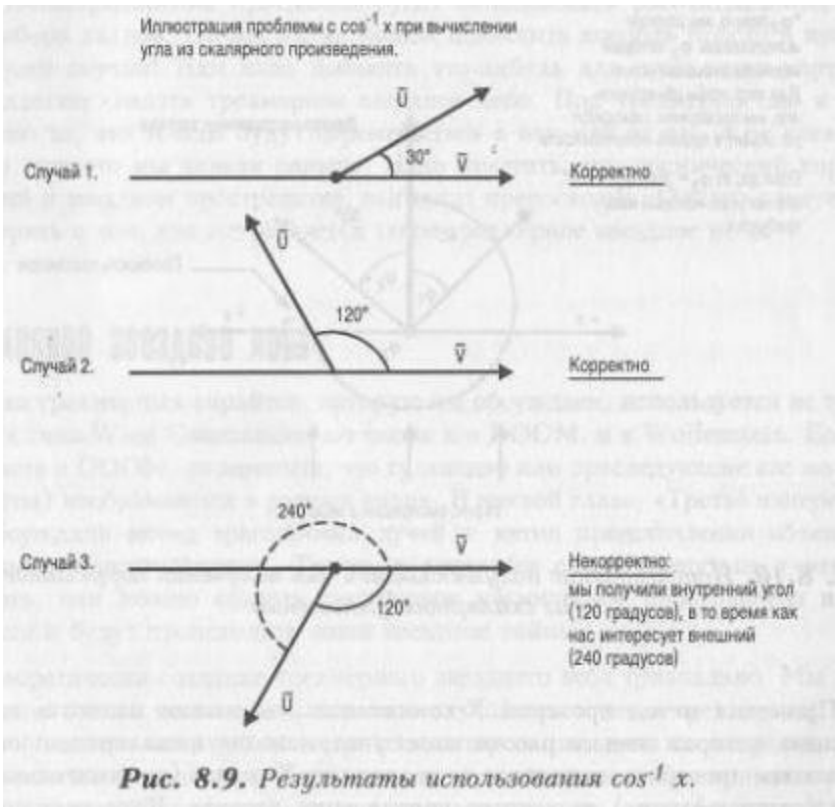


Рис. 8.9. *Результаты использования $\cos^{-1} x$.*

Кстати, формулу 8.4 можно значительно упростить, вспомнив, что нас интересует только плоскость X-Z, так как луч зрения всегда перпендикулярен плоскости просмотра.

Но как же, в конце концов, определить действительный угол? Конечно, вы Могли бы воспользоваться еще и векторным произведением, чтобы решить, корректен ли угол, полученный в результате расчетов по формуле 8.4 или необходимо увеличить его еще на 180 градусов. Однако я слишком не люблю математику (возможно, именно поэтому я и доктор математических наук) и предпочитаю вместо грубой силы использовать тонкую интуицию.

Если мы сообразим, что вектор траектории объекта имеет ту же исходную точку, что и вектор направления взгляда, а затем проверим, в какой из полуплоскостей относительно луча зрения расположен X-компонент вектора траектории, то мы сможем определить, больше или меньше 180° искомый угол. Это наглядно изображено на рисунке 8.10.

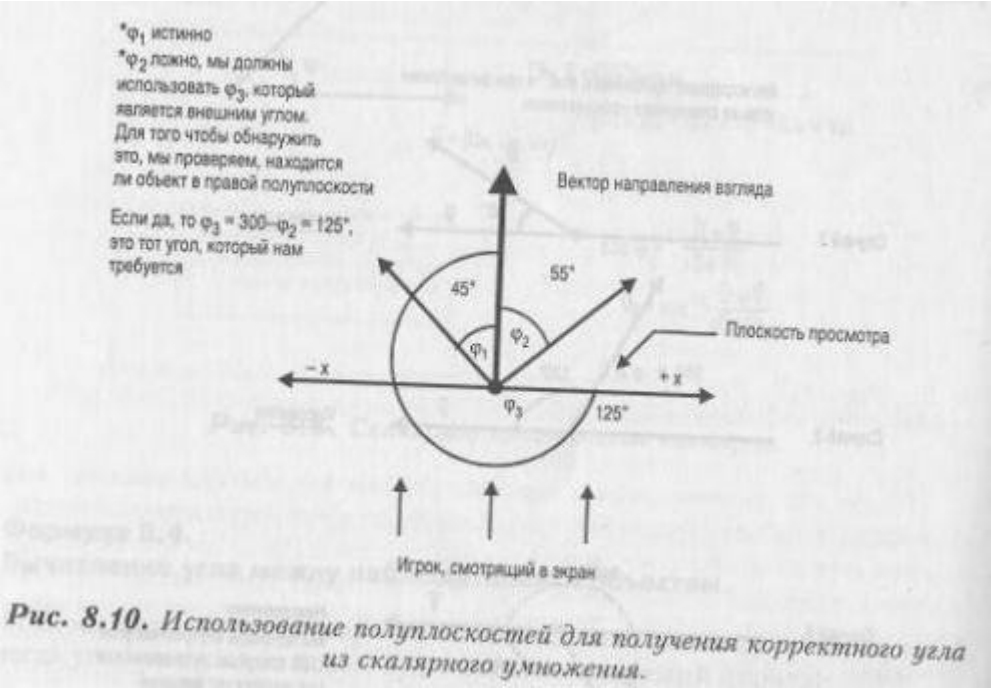


Рис. 8.10. *Использование полуплоскостей для получения корректного угла из скалярного умножения.*

Применяя метод проверки X-компонента, мы можем написать простую функцию, которая вначале рассчитывает угол, используя скалярное произведение, а затем проверяет, находится ли координата X справа (положительная) или слева (отрицательная) от вектора направления взгляда. Если координата X положительная, мы вычитаем угол, полученный с помощью формулы 8.4 из 360 градусов (это все равно, что прибавить 180). Затем мы можем взять рассчитанный угол и разбить его на 12 квадрантов (либо взять его модуль по основанию 12). Полученное число затем можно использовать как индекс для нахождения кадров спрайта. (Конечно, кадры должны быть расположены в правильном порядке, то есть кадрам, полученным при вращении объекта против часовой стрелки с шагом в 30 градусов, должны соответствовать индексы от 0 до 11. При этом нулевой индекс должен указывать на кадр объекта, повернутого тыльной стороной к наблюдателю.)

Если значение координаты X отрицательное, происходит то же самое за исключением того, что будет использован другой банк изображений, и оперировать потребуется с абсолютным значением X.

Кадры, которые я создал для демонстрации этого алгоритма, расположены в файле VRYENTXT.PCX. Они расположены слева направо и сверху вниз. Каждая картинка содержит изображение, повернутое на 30° против часовой стрелки, а в исходной позиции нос корабля направлен прямо в экран (или, с точки зрения игрока, корабль обращен к нему тыльной стороной). Этот же файл мы использовали и в предыдущем примере.

Демонстрационная программа будет использовать рассчитываемые углы для выбора кадров. Но мы же не можем поместить корабль просто в пустоту. Это будет скучно! Нам надо добавить что-нибудь для оживления картинки. Я предлагаю создать трехмерное звездное небо. Под трехмерностью я здесь понимаю то, что звезды будут перемещаться к вам или от вас, а не влево или вправо, как это мы делали раньше, Надо отметить, что космический корабль, летящий в звездном пространстве, выглядит превосходно. Однако следует еще поговорить о том, как же создается такое трехмерное звездное небо.

Трехмерное звездное небо

Техника трехмерных спрайтов, которую мы обсуждаем, используется не только в играх типа Wing Commander, но также и в DOOM, и в Wolfenstein. Если вы поиграете в DOOM, то заметите, что гуляющие или преследующие вас монстры (спрайты) изображаются в разных видах. В шестой главе, «Третье измерение», мы обсуждали метод трассировки лучей и метод представления объектов с помощью многоугольников. Теперь я хотел бы снова вернуться к этому и показать, как можно создать трехмерное космическое пространство внутри которого и будут происходить наши звездные войны.

Теоретически создание трехмерного звездного неба тривиально. Мы могли бы просто задать несколько миллионов случайных точек в трехмерном пространстве, использовать аксонометрическую проекцию для их визуализации и, нарисовать их по пикселям. Этот метод волне применим и обычно используется в симуляторах. Однако у него есть ряд недостатков: он работает медленно, требует много памяти и получаемое в результате изображение быстро надоедает из-за своего однообразия. Но поскольку мы с вами сейчас занимаемся оформлением игры, то должны сделать так, чтобы экран действительно стал трехмерным звездным небом. Но мало сказать, надо еще и сделать! Если вы найдете время и посмотрите Star Trek или подобный симулятор, то увидите звездное небо, созданное с помощью одиночных пикселей (звезд). Эти звезды расположены на экране случайным образом и передвигаются к периферии вначале медленно, а потом все быстрее и быстрее. Это происходит до тех пор, пока звезды не будут отсечены, выйдя за пределы обозримого пространства.

Реализм

Чтобы небо выглядело более реалистично, звезды, находящиеся на меньшем расстоянии от корабля, на котором, мы летим (от нашей виртуальной точки обзора), должны светить ярче. Я предлагаю следующий алгоритм создания трехмерного звездного неба:

1. Для описания звезды создать структуру, содержащую информацию о ее расположении, цвете и скорости перемещения;
2. Создать массив звезд, расположенных по всему экрану и движущихся с разной скоростью;
3. Сделать так, чтобы звезды двигались из центра экрана к краю;
4. Сделать так, чтобы за каждый <?тик» таймера звезда немного ускорялась и становилась ярче;
5. Сделать так, чтобы, достигнув границы экрана, звезда появлялась вновь в иной случайной позиции и перемещалась с другой скоростью;
6. Нарисовать звезды;
7. Вновь вернуться к пункту 4.

Этот метод позволяет получить очень натурально выглядящие звезды и достаточно просто осуществляется. Однако и он тоже не лишен недостатков:

- Во-первых, вы не можете повернуть и должны всегда двигаться вперед;
- Во-вторых, начальная скорость звезды может быть слишком велика по отношению к скорости объекта, летящего по звездному небу.

Первая проблема действительно серьезная. .Если по сценарию игры вам требуется изменять направление полета, то надо воспользоваться каким-нибудь другим алгоритмом. Вторая же проблема, по моему мнению, чисто академическая. Кого волнует, что наш корабль летит со скоростью, во много раз превышающей световую? Это игра, и мы вольны строить такую вселенную, какая нам больше нравится!

Создание звездного неба с использованием целых чисел

Теперь поговорим о том, как можно создать звездное небо используя только целые числа. Мы могли бы для получения более точной картины использовать числа с плавающей запятой, такие, например, как 2,5. Однако мы знаем, что это снижает производительность системы. Нам же требуется создать и визуализировать 50-100 звезд как можно быстрее, чтобы остальное время работы процессора потратить на масштабирование спрайтов и развитие сюжета игры.

Если создавать звездное небо, привлекая только целые числа как значения скоростей звезд, то едва ли вы заметите количественную ошибку, обусловленную заменой чисел с плавающей запятой на целые.

Конечно, мы могли бы принять компромиссный вариант и использовать числа с фиксированной запятой, но в данном случае овчинка выделки не стоит. Я считаю, что при достаточно высоких скоростях звезд потеря дробной части числа несущественна. Наверное, я уже утомил вас, повторяя одно и то же: в компьютерных играх все и всегда должно быть реализовано самым простым и быстрым путем. С другой стороны, такое примитивное на первый взгляд оформление игры, как экран, усыпанный звездами, может отнять у программиста удивительно много времени и сил. Но не забывайте, что отличное оформление - залог того, что игрок погрузится в игру с головой!

Листинг 8.3 содержит полный текст программы, позволяющей перемещать космический корабль по трехмерному звездному небу (отметим, что сам звездолет находится в плоскости X-Z). Для перемещения корабля используется вспомогательная цифровая клавиатура. Клавиши Left и Right вращают корабль, а клавиши Up и Down замедляют и ускоряют его движение. Для выхода из .программы нажмите клавишу Q.

При компоновке исполняемого файла вы должны объединить модуль FINVYREN.C с графической библиотекой GRAPHICS.C.

Листинг 8.3. Последний Шаттл (FINVYREN.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ//////////////////////////////
#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <bios.h>
#include <fcntl.h>
#include <memory.h>
#tinclude <malloc.h>
#include <math.h>
#include <string.h>
#include <graph.h>
#include "grapics.h" // включаем нашу графическую библиотеку
// ПРОТОТИПЫ ////////////////////////////////////
void Create_Scale_Data_X(int scale, int far *row) ;
```

```
void Create_Scale_Data_Y(int scale, int *row);
void Build_Scale_Table(void);
void Scale_Sprite(sprite_ptr sprite,int scale);
void Clear_Double_Buffer(void);
void Timer(int clicks);
void Init_Stars(void) ;
void Move_Stars(void);
void Draw_Stars(void) ;
// ОПРЕДЕЛЕНИЯ ////////////////////////////////////////
#define NUM_STARS      50       // количество звезд на небе
#define MAX_SCALE      200      // максимальный размер спрайта
#define SPRITE_X_SIZE  80       // размеры текстуры спрайта
#define SPRITE_y_SIZE  48
// СТРУКТУРЫ ////////////////////////////////////////// // это звезда
typedef struct star_typ
{
int x,y;        // позиция звезды
int xv,yv;      // скорость звезды
int xa,ya;      // ускорение звезды
int color;      // цвет звезды
int clock;      // число "тиков", которое звезда существует
int acc_time;   // количество "тиков" до ускорения звезды
int acc_count;  // счетчик ускорения
} star, *star_ptr;
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ ////////////////////////////////
unsigned int far *clock = (unsigned int far *)0x0000046CL;
// указатель на внутренний таймер 18.2 "тик"/с
sprite object;
// обобщенный спрайт, который содержит кадры корабля
pcx_picture text_cells;                 // файл PCX с кадрами корабля
int *scale_table_y[MAX_SCALE+l] ;
// таблица предварительно рассчитанных коэффициентов масштабирования
int far *scale_table_x[MAX_SCALE+l];
// таблица предварительно рассчитанных коэффициентов масштабирования
star star_field[NUM_STARSj;             // звездное небо
// ФУНКЦИИ ////////////////////////////
void Timer(int clicks)
{ // эта функция использует внутренний таймер с частотой 18.2 "тик"/с
// 32-битовое значение этого таймера находится по адресу 0000:046Ch
unsigned int now;
// получаем текущее время
now = *clock;
// Ожидаем до истечения указанного периода времени.
// Заметьте, что каждый "тик" имеет длительность примерно 55 мс.
while(abs(*clock - now) < clicks){}
} // конец Timer
////////////////////////////////
void Init_Stars(void)
{
// эта функция инициализирует структуру данных звезд
// при старте программы
int index,divisor;
for (index=0; index<NUM_STARS; index++)
{
star_field[index].x      = 150 + rand() % 20;
star_field[indexl.y      = 90 + rand() % 20;
if (rand()%2==1)
star_field[index].xv     = -4 + -2 * rand() % 3;
else
star_field[index].xv     = 4 + 2 * randO % 3;
if (rand()%2==1)
star_field[index].yv     = -4 + -2 * rand() % 3;
else
star_field[index].yv     = 4 + 2 * randO % 3;
divisor = 1 + rand()%3;
star_field[index].xa    = star_field[index].xv/divisor;
star_field[index] .ya     = star_field [index] .yv/divisor;
```

```
star_field[index].color     = 7;
star_field[index].clock     = 0;
star_field[index].acc_time  = 1 + rand() % 3;
star_field[index].acc_count = 0;
} // конец цикла
} // конец Init Stars
//////////////////////////////////////////////////////
void Move_Stars(void) {
// Эта функция перемещает звезды и проверяет, не вышла ли звезда
// за пределы экрана. Если да, звезда создается вновь.
int index,divisor;
for (index=0; index<NUM STARS; index++)
{
star_field[index].x += star_field[index].xv;
star field[index].y += star_field[index].yv;
// проверка выхода звезды за пределы экрана
if(star_field[index].x>=SCREEN_WJDTH || star_field[index].x<0 ||
star_field[index].y>=SCREEN_HEIGHT || star_field[index].y<0)
{
// восстановление звезды
star_field[index].x      = 150 + rand() % 20;
star_field[index].y      = 90  + randf) % 20;
if (rand()%2==l)
star_field[index].xv     = -4 + -2 * rand() % 3;
else
star_field[index] .xv     = 4 + 2 * rand() % 3;
if (rand()%2==l)
star_field[index].yv     = -4 + -2 * rand() % 3;
else
star_field[index].yv     = 4 + 2 * rand() % 3;
divisor = 1 + rand()%3;
star_field[index].xa     = star_field[index].xv/divisor;
star_field [index] .ya     = star_field.[index] .yv/divisor;
star_field[index].color     = 7;
star_field[index].clock     = 0;
star_field[index].acc_time  = 1 + rand() % 3;
star field[index].ace_count = 0;
} // конец оператора if
//не пора ли ускорить движение звезды
if (++star_field[index].acc_count==star_field[index].acc_time)
{
// обнуляем счетчик
star_field[indexl.acc_count=0;
// ускоряем
star_field[index].xv += star field[index].xa;
star_field[index].yv += star_field[index].ya;
} // конец оператора if
//не пора ли изменить цвет звезды
if (++star_field[index].clock > 5)
{
star_field[index].color = 8;
} // конец оператора if (> 5)
else
if (star_field[index].clock > 10)
{
star_field[index].color =255;
} // конец оператора if (> 10)
else
if (star_field[index].clock> 25)
{
star_field[index].color = 255;
} // конец оператора if (> 25)
} // конец цикла
} // конец Move_Stars
//////////////////////////////////////////////
void Draw_Stars(void)
{
```

```
// эта функция рисует звезды в дублирующем буфере
int index;
for (index=0; index<NUM_STARS; index++)
{
Plot_Pixel_Fast_D(star_field[index].x,star_field[index].y, (unsigned
char)star_field[index].color) ;
} // конец цикла
} // конец Draw_Stars
//////////////////////////////////////////////////////
void Create_Scale_Data_X (int scale, int far *row)
{
// эта функция масштабирует полосу текстуры всех возможных
// размеров и создает огромную таблицу соответствий
int x;
float x_scale_index=0, x_scale_step;
// рассчитываем шаг масштабирования или число исходных пикселей
// для отображения на результирующее изображение за цикл
x_scale_step = (float) (sprite_width)/(float)scale;
x_scale_index+=x_scale_step;
for (x=0; x<scale; x++) {
// помещаем данные в массив для последующего использования
row[x] = (int) (x_scale_index+.5);
if (row[x] > (SPRITE_X_SIZE-1)) row[x] = (SPRITE_X_SI2E-1);
// рассчитываем следующий индекс
x_scale_index+=x_scale_step;
} // конец цикла
} // конец Create Scale_Data X
//////////////////////////////////////////////////////
void Create_Scale_Data_Y(int scale, int *row)
{
// эта функция масштабирует полосу текстуры всех возможных
// размеров и создает огромную таблицу соответствий
int y;
float y_scale_index=0, y_scale step;
// рассчитываем шаг масштабирования или число исходных пикселей
// для отображения на результирующее изображение за цикл
y_scale_step = (float)(sprite_height)/(float)scale;
y_scale_index+=y_scale_step;
for (y=0; y<scale; y++)
{ // помещаем данные в массив для последующего использования
row[y] = ((int)(y_scale_index+.5)) * SPRITE_X_SIZE;
if (row[y] > (SPRITE_Y_SIZE-1)*SPRITE_X_SIZE) row[y]] = (SPRITE_Y_SIZE-1)*SPRITE_X_SIZE;
// рассчитываем следующий индекс
y_scale_index+=y_scale_step;
} // конец цикла
} // конец Create_Scale_Data_Y ////////////////////////////////////
void Build_Scale_Table(void)
{
// эта функция строит таблицу масштабирования путем расчета
// коэффициентов масштабирования для всех возможных размеров
// от 1 до 200 пикселей
int scale;
// резервируем память
for (scale=1; scale<=MAX_SCALE; scale++)
{
scale_table_y[scale] = (int *)malloc(scale*sizeof(int)+1);
scale_table_x[scale] = (int far *)_fmalloc(scale*sizeof(int)+1);
} // конец цикла
// создаем Таблицу масштабирования для осей X и У
for (scale=l; scale<=MAX_SCALE; scale++)
{
// рассчитываем коэффициент для данного масштаба
Create_Scale_Data_Y (scale, (int *)scale_table_y[scale]);
Create_Scale_Data_X (scale, (int far *)scale_table x[scale]);
} // конец цикла
} // конец Build_Scale_Table
//////////////////////////////////////////////////////
```

```
void Scale_Sprite(sprite_ptr sprite,int scale)
{
// эта функция масштабирует спрайт (без отсечения). Масштабирование
// производится с использованием заранее рассчитанной таблицы,
// которая определяет, как будет изменяться каждый вертикальный
// столбец. Затем другая таблица используется для учета
// масштабирования этих столбцов по оси X
char far *work_sprite; // текстура спрайта
int *row_y; // указатель на масштабированные
// по оси У данные (заметьте, что
// это ближний указатель)
int far *row_x;.        // указатель на масштабированные
// по оси X данные (заметьте, что
// это дальний указатель)
unsigned char pixel;    // текущий текстель
int x,                  // рабочие переменные
y,
column, work_offset, video_offset, video_start;
// если объект слишком мал, то и рисовать его не стоит
if (scale<l) return;
// рассчитываем необходимые для масштабирования данные
row_y = scale_table_y[scale];
row_x = scale_table_x[scale];
// выбираем соответствующий кадр спрайта
work_sprite = sprite->frames[sprite->curr_frame];
// рассчитываем начальное смещение
video_start = (sprite->y << 8) + (sprite->y << 6) + sprite->x;
// изображение рисуется слева направо и сверху вниз
for (x=0; x<scale; x++)
{
// пересчитываем адрес следующего столбца
video_offset = video_start + x;
// определяем, какой столбец должен быть отображен,
// исходя из индекса масштабирования по оси X
column = row_x[x];
// Наконец рисуем столбец обычным образом
for (y=0; y<scale; y++)
{
// проверка на "прозрачность"
pixel = work_sprite[work_offset+column];
if (pixel)
double_buffer[video_offset] = pixel;
// индекс следующей строки экрана и смещение в
// области хранения текстуры
video_offset += SCREENJHDTH;
work_offset = row_y[y];
} // конец цикла по Y
} // конец цикла по X
} // конец Scale_Sprite
//////////////////////////////////////////////////////////
void Clear_Double_Buffer(void)
{
// угадали что это?
_fmemset(double_buffer, 0, SCREEN_WIDTH * SCREEN_HEIGHT + 1) ;
} // конец Clear_Double_Buffer
// ОСНОВНАЯ ПРОГРАММА ////////////////////////////////////
void main(void)
{
// корабль помещается в космическое пространство и игрок
// может перемещать его среди звезд
int done=0,                   // флаг выхода
scale=64,
direction=6;              // направление корабля (текущий кадр)
float scale_distance = 24000,
view distance = 256,
// произвольные константы для согласования масштабирования
// плоской текстуры в трассированном пространстве
```

```
x=0,                    // позиция корабля в пространстве
y=0,
z=1024,
xv=0,zv=0,              // скорость корабля в плоскости X-Z
angle=180,
// угол поворота корабля
ship_speed=10;          // величина скорости корабля
// установка видеорежима 320x200x256
_setvideomode(_MRES256COLOR) ;
// все спрайты будут иметь этот размер
sprite_width = 80;
sprite_height = 48;
// создание таблицы соответствия для подсистемы масштабирования
Build_Scale_Table();
// инициализация файла PCX, который содержит все кадры
PCX_Init((pcx_picture_ptr)&text_cells);
// загрузка файла PCX, который содержит все кадры
PCX_Load("vyrentxt.pcx", (pcx_picture_ptr)&text_cells,1) ;
// резервирование памяти для дублирующего буфера
Init_Double_Buffer();
// инициализация звездного неба
Init_Stars() ;
// установка направления и скорости корабля
angle=direction*30+90;
xv = (float)(ship_speed*cos(3.14159*angle/180));
zv = (float)(ship_speed*sin(3.14159*angle/180));
Sprite_Init((sprite_ptr)&object,0,0,0,0,0,0);
// загрузка 12 кадров космического корабля
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells, (sprite_ptr)&object,0,0,0) ;
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells, (sprite_ptr)&object, 1, 1, 0);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,
(sprite_ptr)&object,2,2, 0) ;
PCX_Grap_Bitmap ((pcx_picture_ptr) &text_cells,
(sprite_ptr)Sobject,3,0,1) ;
PCX_Grap_Bitmap((pcx_picture_ptr)&text cells,
(sprite_ptr)&object,4,1,1);
PCX_Grap_Bitmap((pcx_picture_ptr)&text cells,
(sprite_ptr)&object,5,2,1) ;
PCX_Grap_Bitmap ((pcx_picture_ptr) &text_cells,
(sprite_ptr)&object,6,0,2) ;
PCX_Grap_Bitmap ((pcx_picture_pfcr) stext_cells,
(sprite_ptr)&object,7,1,2);
PCX_Grap_Bitmap((pcx picture ptr)&text cells,
(sprite_ptr)Sobject,8,2,2);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,
(sprite_ptr)&object,9,0,3);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_dells,
(sprite_ptr)&object,10,1,3) ;
PCX_Grap_Bitmap ((pcx_picture_ptr) &text_cells,
(sprite_ptr)&object,11,2,3);
// инициализация положения корабля
object.curr_frame = 0;
object.x          = 0;
object.y         =0;
Clear_Double_Buffer() ;
// отслеживание действий игрока и прорисовка корабля
while(!done) {
// нажал ли игрок клавишу?
if (kbhit()) {
switch(getch()) {
case '4' // повернуть корабль влево
{
if (++direction==12)
{
direction==0;
} // конец оператора if
} break;
```

```
case'6': // повернуть корабль вправо
{
if (--direction < 0)
{
direction=11;
} // конец оператора if
} break;
case '8' : // ускорить корабль
{
if (++ship_speed > 20) ship_speed=20;
} break; case '2': // замедлить корабль
{
if (--ship_speed < 0)
ship_speed=0;
} break;
case 'q': // выход из программы
{
done=l;
} break;
default: break;
} // конец оператора switch
// векторы направления и скорости
angle=direction*30+90;
xv = (float)(ship_speed*cos(3.14159*angle/180));
zv = (float)(ship_speed*sin(3.14159*angle/180)};
} // конец оператора if
// переместить корабль
x+=xv;
z+=zv;
// ограничить по ближней плоскости
if (z<256)
Z=256;
// рассчитать размер изображения
scale = (int)( scale_distance/z );
// на основании размера изображения
// рассчитать проекции X и Y
object.x = (int) ((float)x*view_distance / (float)z) + 160 - (scale>>1);
object.y = 100 - (((int) ( (float) y*view_distance / (float)z) + (scale>>1)) );
// ограничить рамками экрана if (object.x < 0)
object.x = 0;
else if (object.x+scale >= SCREEN_WIDTH)
object.x = SCREEN_WIDTH-scale;
if (object.y < 0) object.y = 0;
else
if (object.y+scale >= SCREEN_HEIGHT)
object.y = SCREEN_HEIGHT-scale;
// выбрать нужный кадр
object.curr_frame = direction;
// очистить дублирующий буфер
Clear_Double_Buffer() ;
Move_Stars();
Draw_Stars();
// масштабировать спрайт до нужного размера
Scale_Sprite((sprite_ptr)&object,scale) ;
Show_Double_Buffer(double_buffer);
// вывести на экран информацию для игрока
_settextposition(23,0) ;
printf("Position=(%4.2f,%4.2f,%4.2f)   ",x,y,z) ;
// немного подождать
Timer(1) ;
} // конец while
// удалить файл
PCX PCX_Delete((pcx_picture_ptr)&text_cells) ;
// восстановить текстовый режим
_setvideomode(_DEFAULTMODE);
} // конец функции main
```

Как и прежде, в теле самой программы располагаются только вызовы функций. Это сделано

для облегчения понимания работы алгоритма. При написании реальной, а не демонстрационной программы вам не обязательно поступать так же. Если вы сделаете свою программу чересчур модульной, то может оказаться, что время, потраченное на вызов некоторых функций, станет больше времени их выполнения!

Теперь, после того, как мы узнали математику и механику вывода на экран преобразований трехмерных спрайтов, давайте посмотрим, как я создавал артинки для программ этой главы.

Оцифровка объектов и моделирование

Это самая приятная часть. Я чувствовал себя Джорджем Лукасом, создавая макеты и их изображения для этой главы. Хотя это и крайне увлекательное занятие, оно не столь уж и простое, как многие могли бы подумать. (Это касается тебя, Марк!)

Существует три основных подхода к построению моделей и образов для игр:

- Рисовать их;
- Моделировать их с помощью трехмерных программ типа 3D Studio;
- Оцифровывать кадры, полученные с помощью видеокамеры.

К сожалению, я не профессиональный художник. Хотя я и могу нарисовать свою собственную плоскую картинку, мне слабо нарисовать трехмерный объект под разными углами зрения, не имея перед собой модели. К тому же, я терпеть не могу пользоваться программами моделирования, так как они излишне сложны. Вам потребуется объединить сотни многоугольников, чтобы получить хоть сколько-нибудь реалистичную картинку, да еще и само программное обеспечение стоит очень дорого. Таким образом, для получения трехмерных изображений для этой главы мне оставалось только воспользоваться миниатюрными макетами, видеокамерой и картой для оцифровки.

Теоретически дискретизация миниатюрного макета - довольно простое занятие. Купил видеокамеру или одолжил ее у кого-нибудь, зарядил - и вперед, начинай снимать объект. Это заблуждение! Существует миллион деталей, о которых вы не должны забывать. Освещение, фокус, программное обеспечение, цветовая палитра, разрешение и т. д. - все должно быть сделано, как надо. В своей работе я использовал Creative Labs Video Blaster SE (Special Editions), вместе с Microsoft Video for Windows, Adobe Photostyler и Deluxe Animation фирмы Electronic Art. Кроме этого, я использовал очень дорогую видеокамеру, которую одолжил у своей хорошей знакомой, пообещав упомянуть ее в этой книге. (Диона, я знаю, что на Олимпиаде 1996 года золотая медаль будет твоей!) Наконец, собрав все необходимое, я построил маленькую студию с голубым экраном, вращающейся платформой, освещением и приспособлением для определения угла поворота модели.

Я забыл вам сказать, что найти подходящий игрушечный макет не так легко как кажется. Все вокруг защищено авторскими правами, зарегистрировано или имеет торговую марку. Мой вам совет, купите 20 разных макетов кораблей, судов, космических шлюпок и из всего этого постройте свой собственный сверхъестественный объект. Иначе можно и в суд угодить!

Я хочу рассмотреть основные моменты создания своей студии, чтобы помочь вам безболезненно разрешить все проблемы, возникающие при оцифровке собственного игрового объекта.

Создание съемочного мини-павильона

Вам необходимо:

- Во-первых, чистое, просторное место, чтобы установить видеокамеру и подставку. Расстояние между ними должно быть не менее четырех шагов. Это позволит вам поэкспериментировать с фокусом;
- Во-вторых, я считаю, что комната для съемок должна быть по возможности одноцветной. Я не советую вам снимать в комнате с розовыми и голубыми стенами. Больше всего в данной ситуации подойдут белые стены и темный потолок;
- Затем сделайте небольшую площадку для фотографии, фон которой вы сможете менять так, как это показано на рисунке 8.11. Я вырезал два кусочка оргстекла и установил их под прямым углом друг к другу. Затем я использовал цветную бумагу, для создания пола и фона для объекта.

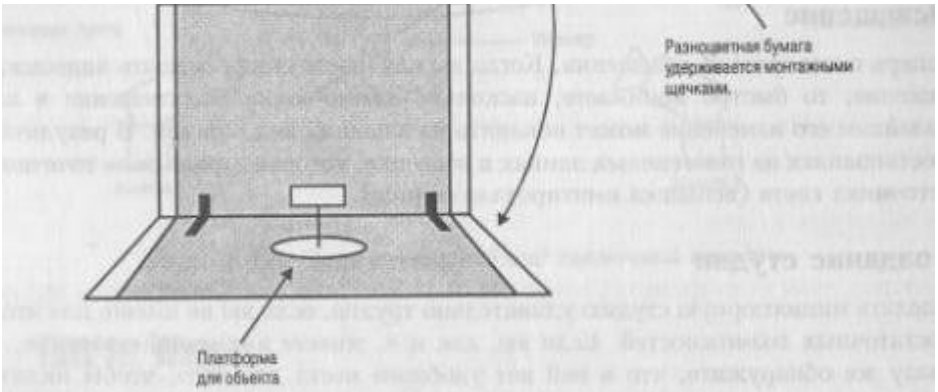


Рис. 8.11. *Съемочная студия, позволяющая легко изменять цвет фона.*

Фон

В кино используется голубой фон, что позволяет отфильтровать синие тона в изображении. Хотя этот метод широко известен, у меня возникли с ним проблемы. В конце концов, я вынужден был использовать черный экран. Синий же фон имеет смысл использовать, если вы будете снимать черные объекты. Затем я подбирал освещение до тех пор, пока изображение объекта не получилось максимально контрастным по отношению к фону.

Платформа

Раз уж у вас есть камера и импровизированная недорогая студия, то вы должны сделать для своего объекта подходящую платформу. Она не должна портить изображение и впоследствии легко должна удаляться из полученной картинки. Вначале я использовал прозрачную подставку из оргстекла, но быстро убедился, что отражения на ее поверхности создают причудливые световые эффекты. Тогда я приклеил с нижней стороны подставки бумагу. Затем я проделал маленькое отверстие снизу моей модельки и закрепил ее на платформе. Получилось совсем неплохо. Правда, я мог поворачивать модель только вокруг одной оси и не мог наклонить ее. Однако, меня это вполне устраивало, так как мне и не требовались такие сложные ракурсы.

Освещение

Теперь поговорим об освещении. Когда вы начинаете оцифровывать видеоизображение, то быстро понимаете, насколько важно хорошее освещение и как малейшее его изменение может повлиять на внешний вид объекта. В результате я остановился на галогеновых лампах и вспышке, которая играла роль точечного источника света (вспышка имитировала солнце).

Создание студии

Сделать миниатюрную студию удивительно трудно, если вы не имеете для этого остаточных возможностей. Если вы, как и я, живете в обычной квартире, то сразу же обнаружите, что в ней нет удобного места для того, чтобы пилить, сверлить или рисовать. Хотя, как я уже говорил выше, мне хватило для создания студии лишь нескольких листов оргстекла, соединенных под прямым углом.

Если "вы не очень любите мастерить, можете пойти и купить несколько листов белой, черной и синей конструкторской бумаги. Я думаю, что стоит поэкспериментировать со всеми тремя цветами и выбрать для себя лучший. Приклейте бумагу к стене и поставьте вашу модельку перед ней на пол на какую-нибудь подставку. Для начала это вполне подойдет, но будет немного неудобно, потому что вам придется много ползать по полу на коленях.

Я думаю, что окончательно студия должна выглядеть так, как это показано на рисунке 8.12. Это должен быть стол того же цвета, что и параболически изогнутый; экран фона. Вращающийся экран должен быть закреплен на шарнирах, чтобы вы могли изменять его наклон и положение вместе с наклоном и положением объекта. Более того, желательно иметь две камеры, расположенные под углом друг к другу, чтобы снимать объект одновременно в двух разных ракурсах.

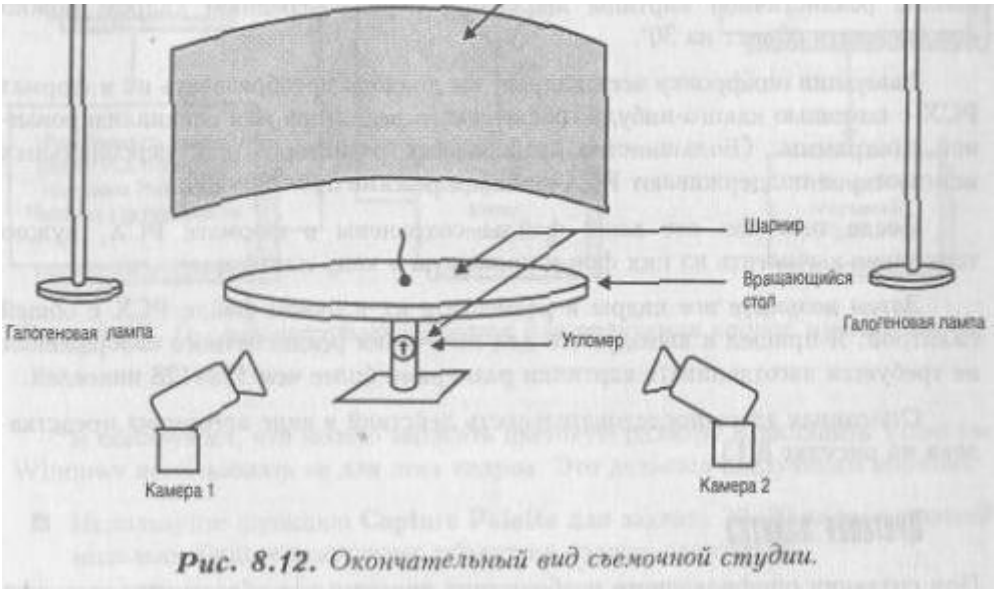


Рис. 8.12. Окончательный вид съемочной студии.

Video for Windows

Когда все ваши технические проблемы решены и вы получили приличное видеоизображение вашего макета, настало время использовать компьютера программное обеспечение (или, скорее, я бы сказал, его отсутствие) для того, чтобы перевести его в цифровую последовательность. Я сам не люблю понапрасну тратить много денег, и поэтому не буду и вас убеждать приобрести оборудование и программное обеспечение на сотни или тысячи долларов. Следовательно, нам придется использовать то, что поставляется с платой ввода графической информации. Я имею в виду Microsoft Video for Windows, которая, в общем-то, предназначена, скорее, для оцифровки сюжетов, а не для моментальных снимков вроде наших.

Так как почти все платы ввода графической информации продаются вместе с Microsoft Video for Windows, я хочу рассказать, как использовать Video for Windows для обработки образов ваших игровых объектов.

- При создании каждого кадра используйте опцию single-frame (одиночный кадр);
- Создав кадр, сохраняйте его в формате BMP-файла, а затем с помощью другой программы преобразуйте его в PCX-формат.
- При создании кадров с вращающимся объектом расположите под ним круглый транспортир, чтобы точно повернуть объект на нужное количество градусов. Затем, когда вы оцифруете все ваши образы, запишите их в файлы с именами namexxx.bmp, где xxx - угол поворота. Я считаю, что для получения вполне реалистичной картины движения, перед очередным кадром можно поворачивать объект на 30°.
- Завершив оцифровку всех кадров, вы должны преобразовать их в формат PCX с помощью какого-нибудь графического редактора или специализированной программы. (Большинство графических редакторов для персональных компьютеров поддерживают PCX-файлы в режиме 320x200x256.)
- После того как все ваши файлы сохранены в формате PCX, нужно тщательно вычистить из них фон и попавшую в кадр платформу.
- Затем возьмите все кадры и разместите их в одном файле PCX с общей палитрой. Я пришел к выводу, что для получения реалистичного изображения не требуется изготавливать картинки размерами более чем 128x128 пикселей.

Описанная здесь последовательность действий в виде алгоритма представлена на рисунке 8.13.

Цветовая палитра

При создании оцифрованного изображения программное обеспечение пытается определить оптимальный набор цветов для каждого из сканируемых кадров в отдельности. Это не подходит для нашего случая, так как у нас должна быть одна и та же палитра для всех кадров. При желании мы, конечно, могли бы пользоваться несколькими палитрами, но тогда пришлось бы менять их каждый раз при смене ракурса объекта. А это представляет определенную сложность.

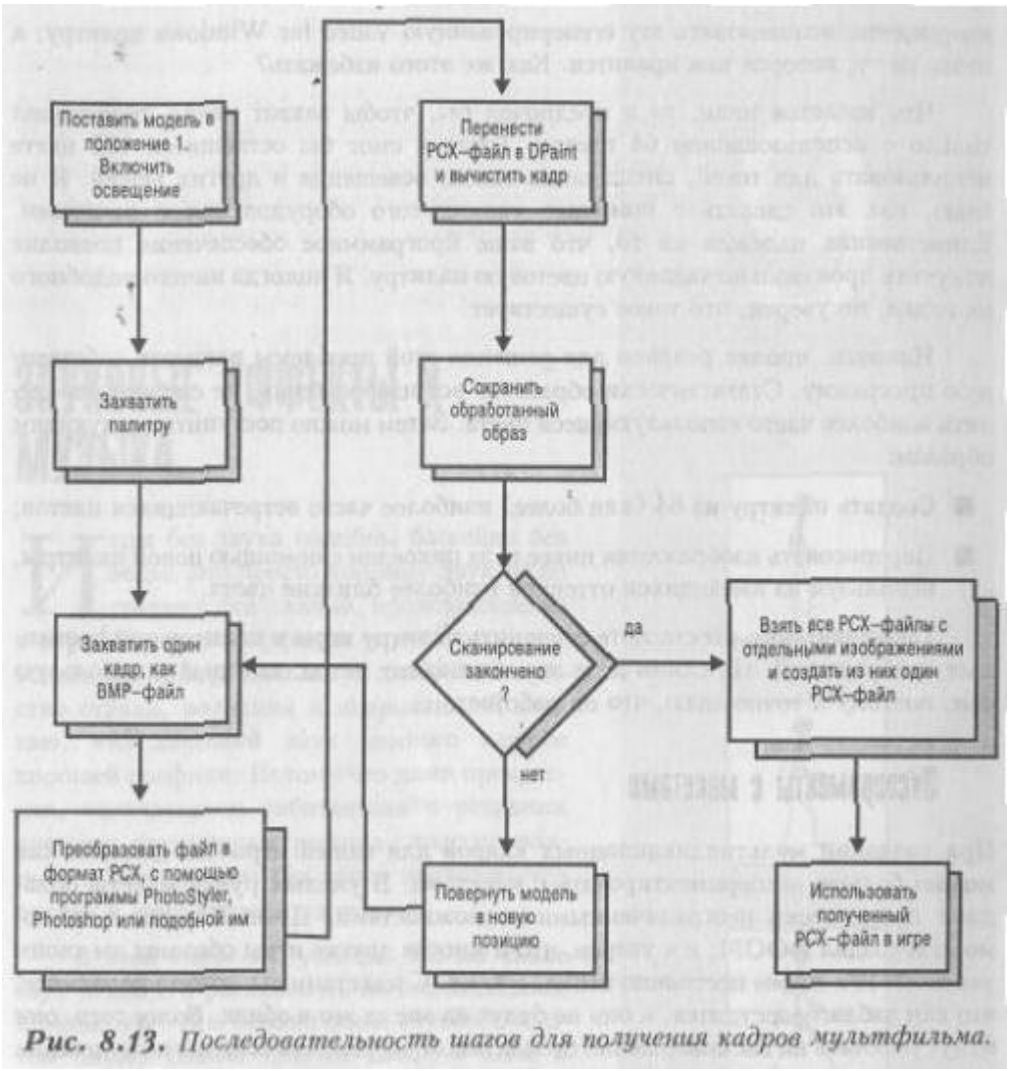


Рис. 8.13. *Последовательность шагов для получения кадров мультфильма.*

Я обнаружил, что можно записать цветовую палитру и заставить Video for Windows использовать ее для всех кадров. Это делается следующим образом:

- Используйте функцию Capture Palette для захвата 20-30 кадров с вашей моделью. Будьте аккуратны, объект не должен сдвигаться!
- Используя эти 20-30 кадров, программа усреднит палитры каждого изображения и подберет оптимальную. Затем вы должны будете использовать эту палитру для всей последовательности кадров.

Все это замечательно, но в данном случае окажутся задействованными все 256 цветов палитры. В результате все прочие изображения в вашей игре будут

вынуждены использовать эту сгенерированную Video for Windows палитру, а вовсе не ту, которая вам нравится. Как же этого избежать?

Что касается меня, то я предпочел бы, чтобы захват кадра, происходил только с использованием 64 цветов. Тогда я смог бы оставшиеся 192 цвета использовать для теней, смешивания тонов, освещения и других вещей. Я не знаю, как это сделать с помощью упомянутого оборудования и программ. Единственная надежда на то, что ваше программное обеспечение позволит получить произвольно заданную цветовую палитру. Я никогда ничего подобного не видел, но уверен, что такое существует.

Наконец, вполне реально для решения этой проблемы написать собственную программу. Статистически обработав все изображения, не сложно определить наиболее часто использующиеся цвета. Затем можно поступить следующим образом:

- Создать палитру из 64 (или более) наиболее часто встречающихся цветой;
- Перерисовать изображения пиксель за пикселем с помощью новой палитры, используя из имеющихся оттенков наиболее близкие цвета.

Таким образом, вы сможете соединить палитру игры и палитру оцифрованных изображений. На самом деле это именно тот метод, который я использую сам, поэтому я точно знаю, что он работает.

Эксперименты с макетами

При создании мультипликационных кадров для вашей игры вы должны как можно больше экспериментировать с макетами. В умелых руках макеты обладают практически неограниченными возможностями. Джон Кармак с их помощью создал DOOM, и я уверен, что и многие другие игры обязаны им своим успехом. И я также постоянно использую их. С макетами вы можете делать все, что вам заблагорассудится, и они не будут на вас за это в обиде. Более того, они будут работать на вас совершенно бесплатно! Я уверен, что если вы не художник и собрались сами создать

компьютерную игру, то для создания реалистичныизображений макеты — это наилучшее решение.

Коротко! (То есть итог)

В этой главе мы узнали кое-что новое, но главное, научились делать сложные и трудные вещи, используя самые простые методы. Мы разобрались, как масштабировать спрайты и проецировать их на экран, используя аксонометрические преобразования. Мы снова поговорили об отсечении. И, наконец, мы окунулись в восхитительный мир производства компьютерных игр с использованием настоящих макетов.

ЗВУКОВЫЕ ЭФФЕКТЫ И МУЗЫКА

Игры без звука подобны бассейну без воды. Это просто нонсенс! Звук тесно связан с действиями, происходящими в игре. Звуковые эффекты и музыка - это основа магии игр. Они могут изобразить чувство страха, волнения и неприязни. Я считаю, что хороший звук намного важнее хорошей графики. Потому что даже прекрасная, замечательно работающая в реальном времени, трехмерная графика с трассированным пространством без звука превращаетсяпросто в двигающиеся картинки. В этой главе мы узнаем, что же такое Звук и как его реализовать на персональном компьютере. Мы ограничимся звуковой картой Sound Blaster фирмы Creative Labs и совместимыми с ней устройствами. В этой главе мы рассмотрим:

- Основы звука;
- Устройство и архитектуру звуковой карты Sound Blaster;
- Создание и воспроизведение оцифрованных звуковых эффектов;
- Частотный синтез звука;
- MIDI-музыку;
- Специальные методы работы со звуком.
- Основы звука

Со звуком мы сталкиваемся каждый день. Это одно из понятий, определить которое не очень тяжело. Звук - это человеческое восприятие волн давления, распространяющихся в воздушной среде, точно так же, как свет - восприятие электромагнитных волн, распространяющихся в пространстве. Вокруг звучащего объекта воздух расширяется и сжимается. Это расширение и сжатие порождает волны, которые, в конце концов, и достигают вашего уха, создавая переменное давление на барабанные перепонки. На рисунке 9.1 изображен источник звука и распространение звуковых волн от него.

Рис. 9.1. *Источник звука и распространение звуковых волн.*

Распространение звуковых волн из одной точки в другую по воздуху является механическим, так как оно происходит за счет передачи молекулами воздуха своей кинетической энергии друг другу. Когда волна распространяется, она теряет свою энергию из-за трения (мы воспринимаем это как снижение уровня громкости) до тех пор, пока, наконец, ее энергия окончательно не поглотится воздухом.

Звуковые волны имеют некоторые интересные свойства. Например, известный эффект Доплера. Все мы знаем разницу между звуком приближающегося и удаляющегося поезда. Данный эффект обусловлен тем, что скорость распространения звуковых волн зависит от скорости самого поезда. Не забывайте, что в отличие от световых волн, звук распространяется медленно. В среднем их скорость составляет 600 миль в час, так что поезд, двигающийся со скоростью 60 миль в час, может в достаточной степени повлиять на распространение звука. Если вы вспомните, звук приближающегося поезда выше и его тон (или частота колебаний) постепенно возрастает. После того как поезд прошел, тон (частота) снижается. Все это происходит из-за того, что передвигающийся объект порождает фронт звуковой волны в результате локального повышения давления. Таким образом, поезд одновременно и передвигается, и порождает звук, вследствие чего частота звука приближающегося и удаляющегося поезда различаются.

Теперь поговорим о специфических терминах, которые мы будем использовать при рассмотрении различных звуковых эффектов. Для наглядного представления звуков обычно используют двухмерные графики (см. рис. 9.2).

Рис. 9.2. *Форма звуковой волны.*

По оси X обычно откладывают время, а по оси Y — амплитуду звука. Чистая нота ДО первой октавы (440Гц) будет выглядеть так, как это показано на рисунке 9.3.

Как вы можете видеть, волна порождает сама себя снова и снова и внешне выглядит как функция от времени. Это одно из свойств чистого тона; он может быть описан функцией синуса. Математически это записывается так:

```
F(t)=sin(2*Pi*F*t)
```

где F — генерируемая нами частота.

Как легко увидеть на рисунке 9.3, из формы волны мы можем получить несколько параметров. Один из них - *амплитуда* (или, проще говоря, уровень громкости). Ее обычно измеряют в *децибелах.* Децибелы изменяются по логарифмической шкале: звук в 5дБ в 10 раз сильнее звука в 4дБ. (По такой же шкале измеряют силу землетрясения. При 5 баллах лишь немного подрагивает, а при 7 уже рушатся горы, ведь такое землетрясение в 100 раз сильнее!)

Другая величина, определяемая по графику, — это длина звуковой волны. *Длина волны* определяется, как время или расстояние между двумя вершинами синусоиды. Легко видеть что это длительность одного полного цикла звука. Посмотрите для примера на рисунок 9.4. Здесь изображены два разных волновых графика.

Частота одной волны - 500Гц, а другой - 1000Гц. Тысячегерцовая волна имеет в два раза больше полных циклов за единицу времени, чем пятисотгерцовая.

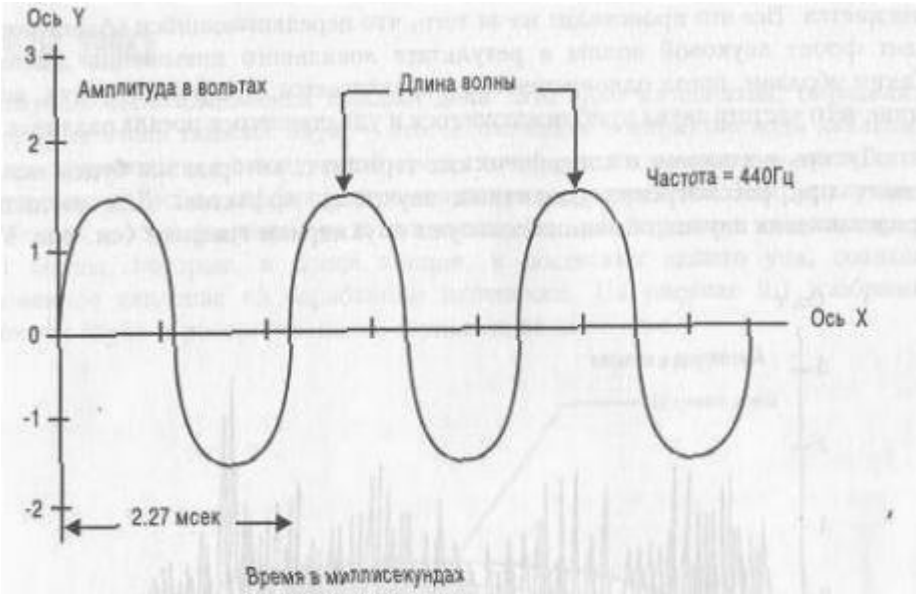


Рис. 9.3. *Представление ноты ДО первой октавы (440Гц).*

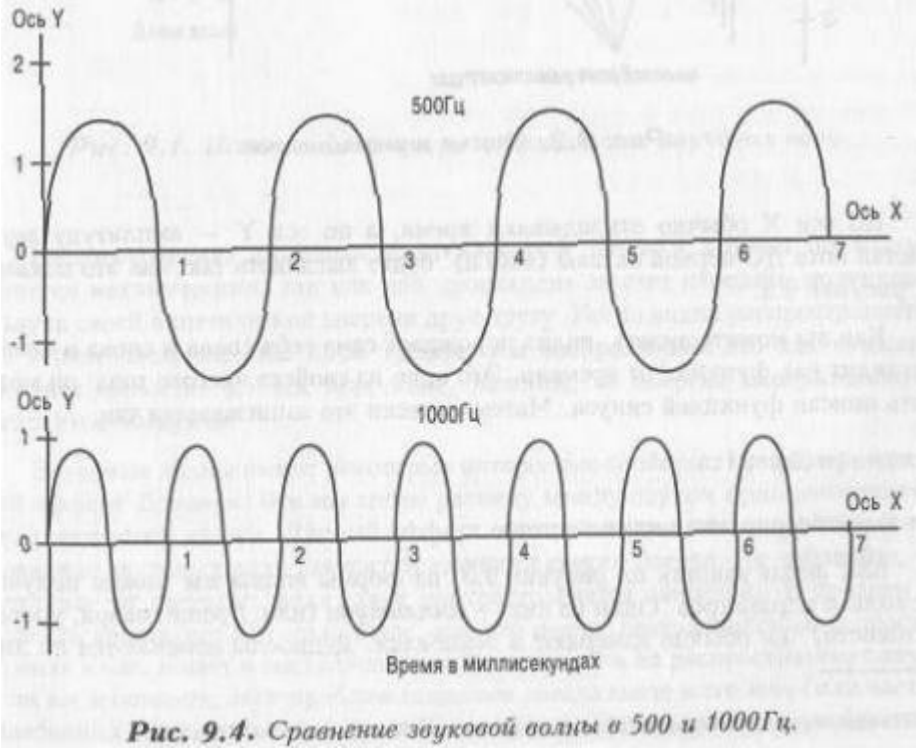


Рис. 9.4. *Сравнение звуковой волны в 500 и 1000Гц.*

Теперь мы должны поговорить о *частотной характеристике восприятия (frequency response).* Обычно человек воспринимает звук, как показано на рисунке 9.5.

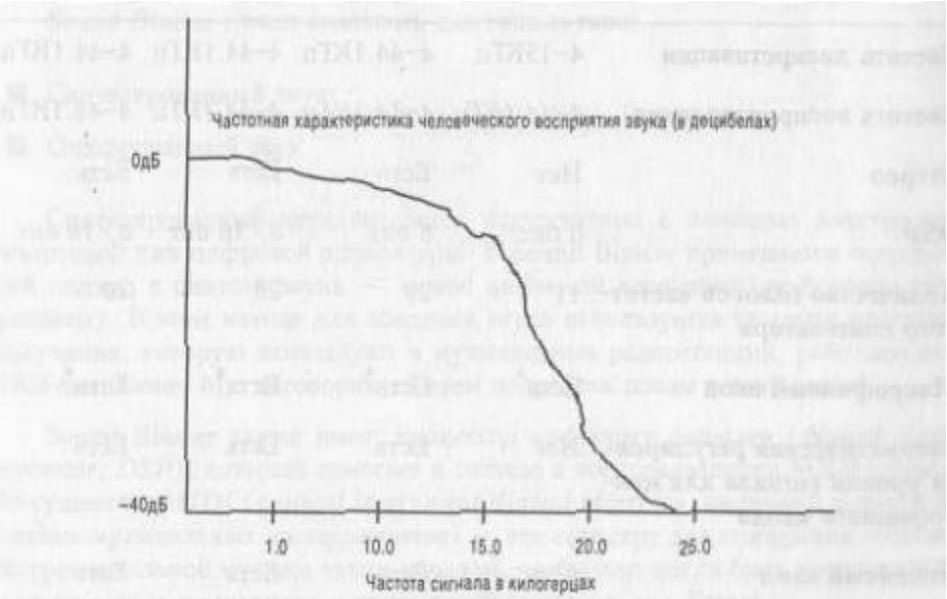


Рис. 9.5. *Частотная характеристика человеческого восприятия звука.*

Мы слышим в диапазоне 20Гц-15КГц, и восприимчивость обычно сильно падает к 20КГц (большинство людей вообще не воспринимает звук за этим порогом). Следовательно, мы должны создавать наше звуковое сопровождение игры именно в таких пределах.

Достаточно о том, что такое звук. Попробуем его сделать!

Устройство и архитектура звуковой карты Sound Blaster

Sound Blaster — сложная звуковая карта с большими возможностями. Управлять ею не так уж и просто, и мы наверняка не сможем рассмотреть все ее впечатляющие способности в одном-единственном разделе. Поэтому мы приведем здесь только основные характеристики современных моделей Sound Blaster и кратко остановимся на их функциональных различиях.

На данный момент существует четыре различных модификации Sound Blaster. Их характеристики приведены в таблице 9.1.

Частота воспроизведения	4–44.1КГц	4–44.1КГц	4–44.1КГц	4–44.1КГц
Стерео	Нет	Есть	Есть	Есть
DSP	8 бит	8 бит	8/16 бит	8/16 бит
Количество голосов частотного синтезатора	11	20	20	20
Микрофонный вход	Есть*	Есть*	Есть*	Есть*
Автоматическая регулировка уровня сигнала для микрофонного входа	Нет	Есть	Есть	Есть
Линейный вход	Есть	Есть	Есть	Есть
Выход для акустических систем	Есть	Есть	Есть	Есть
Линейный выход	Нет	Есть**	Есть**	Есть**
Вход для проигрывателя компакт-дисков	Нет	Есть	Есть	Есть
Интерфейс CD-ROM	Нет	Есть	Есть	Есть

Функции	Sound Blaster V2.0	Sound Blaster Pro V2.0	Sound Blaster 16	Sound Blaster 16 ASP
Микшер	Нет	Есть	Есть	Есть
Регулировка тембра	Нет	Нет	Есть	Есть
Распознавание голоса	Нет	Нет	Нет	Есть
Аппаратное сжатие звука	Нет	Нет	Нет	Есть

* моно

** внутренний

Sound Blaster может создавать два типа звуков:

- Синтезированный звук;
- Оцифрованный звук.

Синтезированный звук создается искусственно с помощью электронной аналоговой или цифровой аппаратуры. В Sound Blaster применяется современный подход к синтезу звука — *метод цифровой частотной модуляции (FM synthesis).* В этом методе для создания звука используется та самая частотная модуляция, которую используют и музыкальные радиостанции, работающие в УКВ-диапазоне. Мы поговорим об этом подробнее позже в этой главе.

Sound Blaster также имеет процессор *цифровых сигналов (digital signal processor, DSP),* который помогает в синтезе и воспроизведении MIDI-музыки. По-существу, *MIDI (musical instrument digital interface, цифровой интерфейс электромузыкальных инструментов)* — это стандарт для оцифровки голоса и инструментальной музыки таким образом, чтобы они могли быть воспроизведены с помощью компьютера или такого синтезатора, как Yamaha.

Кроме синтеза звука Sound Blaster позволяет оцифровывать; а затем воспроизводить такие звуковые фрагменты, как речь или различные эффекты. Это очень полезное свойство, так как некоторые звуки очень сложно или даже невозможно создать только с помощью частотного синтезатора и DSP. В наших играх мы как раз и будем использовать оцифрованные звуковые эффекты вместе с MIDI-музыкой.

В следующем разделе мы рассмотрим как Sound Blaster работает с оцифрованным звуком.

Оцифрованный звук

Итак, начнем! Моя первая попытка оцифровать звук для проигрывания на Sound Blaster была подобна открытию нового континента. Получение именно такого звука, какой вы хотели услышать - это большая победа! Гармония звука и сюжета улучшает игру на порядок. Но для того, чтобы этого достичь, надо вначале разобраться, что же такое оцифрованный звук.

Посмотрите на рисунок 9.6. Это графическое изображение того, как я произношу: «Что случилось?» Как вы можете заметить, этот график сильно отличается от графиков чистых тонов, которые мы рассматривали ранее. Спрашивается, откуда же такая разница? Ответ прост: «Это реальный человеческий голос».

Рис. 9.6. Пример оцифровки звука.

Человеческий голос богат и сложен. Это следствие того, что кроме основного тона в нем присутствуют и гармоники. Когда человек говорит, то в каждый момент времени звук его речи состоит из многочисленных обертонов основной частоты. Чтобы оцифровать эту информацию мы

должны сделать две вещи:

- Преобразовать информацию в электронный сигнал;
- С постоянной частотой дискретизировать этот сигнал.

Во-первых, мы должны конвертировать звук в форму, которую сможем обрабатывать (то есть в цифровую). Это делается с помощью аналогово-цифрового преобразователя- Он преобразует аналоговый сигнал в цепочку цифровых импульсов, состоящих из восьми или шестнадцати битов. Фактически, это означает, что сигнал превращается в последовательность 8- или 16-битных чисел. При использовании восьми битов сигнал может быть представлен числами в - диапазоне от -127 до +128. А если мы используем 16 битов, то диапазон расширяется от -32767 до +32768, что, естественно, дает более высокое качество.

Затем мы должны дискретизировать сигнал с постоянной частотой. Например, представим, что мы дискретизируем разговор человека с частотой 8КГц, используя восемь битов, на протяжении 10 секунд. Это займет 80 кило6айт!!! Как вы;можете видеть, оцифрованный звук требует очень много памяти, вот почему такое большое значение имеют частотные синтезаторы.

Вы можете возразить: «Ну так давайте понизим частоту оцифровки». Это, конечно, можно сделать, но тогда в результате мы потеряем часть данных о звуке вследствие так называемого шума квантования. Пусть наш сигнал имеет частоту 10КГц и мы дискретизируем его с частотой 6КГц. В результате мы не сможем точно воспроизвести форму волны. Практически, а точнее, не столько практически, сколько в полном соответствии со строгим законом Шеннона, вы должны дискретизировать сигнал с частотой в два раза большей, чем его максимальная частота. Только тогда вы сможете затем точно воспроизвести оцифрованный сигнал. Это значит, что если вы хотите точно воспроизвести звук или человеческую речь с частотным диапазоном приблизительно до 20КГц, вы должны дискретизировать его с частотой в два раза большей, то есть 40КГц. Проигрыватели компакт-дисков как раз и работают с частотой 44,1 КГц. На такой скорости вы не потеряете ничего! Звуковые эффекты DOOM, на мой взгляд, оцифрованы где-то на частоте 11 КГц, что являет собой «золотую середину».

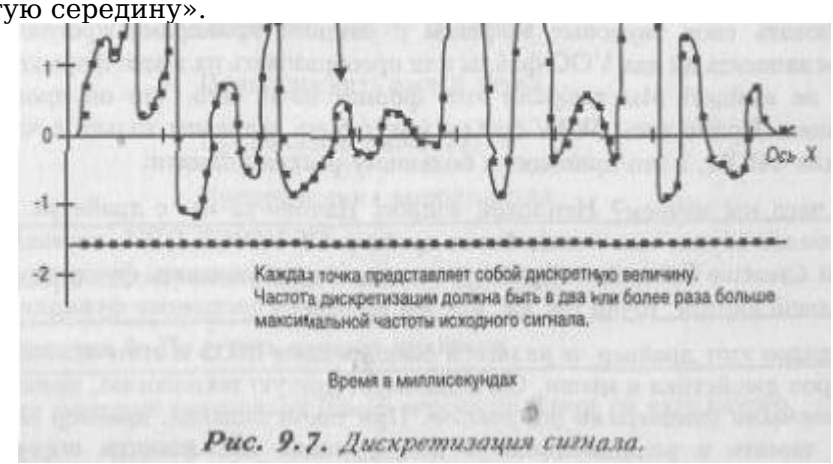


Рис. 9.7. Дискретизация сигнала.

Мы определенно не можем записывать и воспроизводить звуки на такой скорости. Оцифровка, сравнимая по качеству с CD требует слишком много памяти. Для наших целей нам вполне достаточно 6КГц и восьми бит.

На рисунке 9.7 изображен процесс дискретизирования сигнала.

Воспроизведение оцифрованного звука

Теперь поговорим о воспроизведении оцифрованных звуков. Я не буду показывать вам, как оцифровывать звуки. Вы сами можете воспользоваться одним из десятков программных пакетов, предназначенных для этой цели. Тем более, что в самой компьютерной игре звук не надо записывать, его надо воспроизводить!

Из этих соображений я снабдил эту главу условно-бесплатной программой Blaster Master. Эта программа работает в среде MS-DOS и позволяет делать с оцифрованным звуком все что угодно. Она может записывать звук в файлы различных форматов и применять к звуку специальные эффекты (эхо, реверберацию, смену тональности и т. д.)

Примечание

Оцифрованный звук, как и любая другая информация, должен иметь определенный формат для хранения данных. На персональных компьютерах наиболее распространены форматы WAV и VOC. Оба из них, кроме собственно звуковых данных, имеют специальные заголовки. Формат WAV был предложен в Windows, а формат VOC является стандартом «де-факто».

Blaster Master способен преобразовывать данные из формата VOC в формат WAV и обратно. Демонстрационные игры, которые мы будем рассматривать в этой книге, используют исключительно формат VOC. Поэтому, прежде чем использовать свои звуковые эффекты с нашими примерами программ, вы должны записать их как VOC-файлы или преобразовать их в этот формат, иначе ничего не выйдет! Мы выбрали этот формат из-за того, что он проще для понимания. Кроме

того, WAV-файлы могут быть записаны только с частотой 11,22 или 44КГц, а это приводит к большому расходу памяти.

С чего мы начнем? Неплохой вопрос. Начнем-ка мы с драйвера. Чтобы проигрывать звуки, нам понадобится драйвер CT-VOICE.DRV, поставляемый фирмой Creative Labs. Этот драйвер позволяет нам вызывать функции работы со звуковой картой, точно так же, как мы вызываем системные функции BIOS.

Однако этот драйвер не является расширением BIOS и этим отличается от драйверов джойстика и мыши. Он использует другую технологию, называемую загружаемыми драйверами устройств. При таком подходе, драйвер загружается в память и реализованные в нем функции исполняются посредством передачи управления по определенному смещению относительно начала кода драйвера. Передача параметров функциям драйвера осуществляется загрузкой их значений в определенные регистры процессора.

(Впервые я столкнулся с этой технологией около семи лет назад. И хотя я уже долгое время занимался программированием, не сразу сообразил, что мне надо самостоятельно загрузить драйвер в память, а затем передать ему управление.)

CT-VOICE.DRV имеет множество команд, и я не буду детально обсуждать каждую из них. Поговорим только о тех, которые потребуются нам для загрузки и воспроизведения оцифрованных звуков. В таблице 9.2 приведены необходи мые нам функции драйвера.

Таблица 9.2. Подмножество функций драйвера CT-VOICE.DRV.

Функция	**Описание**
0	Получить версию драйвера
1	Установить базовый адрес ввода/вывода
2	Установить прерывание DMA
3	Инициализировать драйвер
4	Включить/выключить динамик
5	Установить адрес слова состояния
6	Начать вывод звука
8	Остановить вывод звука
9	Закончить работу драйвера
10	Приостановить вывод звука
11	Продолжить вывод звука
12	Прервать цикл вывода звука

Теперь поговорим о каждой из этих функций в деталях.

Функция 0: Получить версию драйвера

Эта функция возвращает номер версии драйвера от него самого.

Вход: BX = 0

Выход: AH =старшая часть номера версии.
AL = младшая часть номера версии.

Функция 1: Установить базовый адрес ввода/вывода

Функция устанавливает базовый адрес ввода/вывода, используемый драйвером. Вы должны задать адрес, который соответствует установленному для платы порту ввода/вывода. Если вы не вызываете эту функцию, драйвер по умолчанию использует адрес 220h.

Вход: BX = 1

AX = базовый адрес ввода/вывода.

Выход: Ничего.

Функция 2: Установить прерывание DMA

Функция устанавливает номер прерывания драйвера прямого доступа к памяти, который используется Sound Blaster для сообщения о завершении передачи данных.

Вход: BX=2

AX = Номер прерывания драйвера прямого доступа к памяти.

Выход: Ничего.

Функция 3: Инициализировать драйвер

Эта функция инициализирует и начинает работу драйвера.

Вход: BX=3

Выход: AX = 0 - драйвер успешно инициализирован;

= 1 - некорректная версия драйвера;

= 2 -ошибка операции чтения/записи устройством ввода/вывода;

= 3 - ошибка прерывания драйвера прямого доступа к памяти.

Функция 4: Включить/выключить динамик (только для базовой модели Sound Blaster)

Функция включает или выключает вывод звука.

Вход: BX = 4

AL = 0 - выключить;

= 1 - включить.

Выход: Ничего.

Функция 5: Установить адрес слова состояния

Функция сообщает драйверу адрес переменной, в которой будет храниться информация О состоянии. Вы определяете переменную и затем передаете функции ее сегмент и смещение, таким образом, драйвер получает возможность сообщать вашей программе важную информацию о своих параметрах состояния.

Вход: BX=5

ES:DI = сегмент и смещение переменной, которую вы хотите использовать для хранения слова состояния.

Выход: Ничего.

Функция 6: Начать вывод звука

Это та самая функция, которая все и делает. Она проигрывает VOC-файл, используя прямой доступ к памяти. Однако мы должны передать ей указатель на звуковые данные.

Вход: BX = 6

ES:DI = сегмент и смещение VOC-файла, который вы хотите проигрывать; учтите, что надо быть внимательным и передать функции адрес начала собственно звуковых данных (то есть пропустить информационный заголовок файла; подробнее мы это рассмотрим ниже).

Выход: AX = 0 - успешное выполнение операции, в противном случае возвращается ненулевое значение.

Функция 8: Остановить вывод звука

Функция останавливает все операции ввода/вывода и сбрасывает переменную CT_VOICE_STATUS в ноль.

Вход: BX=8

Выход: Ничего.

Функция 9: Закончить работу драйвера

Функция заканчивает работу драйвера. Однако она не выгружает драйвер из памяти - вы должны сделать это самостоятельно.

Вход: BX=9

Выход: Ничего,

Функция 10: Приостановить вывод звука

Функция приостанавливает вывод звука. Вы можете вновь запустить звук, использовав функцию 11 «Продолжить вывод звука».

Вход: BX=10

Выход: AX = 0 - успешное выполнение операции;

= 1 - вывод звука не происходил.

Функция 11: Продолжить вывод звука

Функция продолжает приостановленный вывод звука.

Вход: BX=11

Выход; AX = 0 - успешное выполнение операции;
= 1 - вывод звука не прерывался.

Функция 12: Прервать цикл вывода звука

Функция прерывает цикл вывода оцифрованных данных.

Вход: BX = 12
AX = 0 - прервать цикл в конце текущей итерации;
= 1 - прервать цикл немедленно.

Выход: AX = 0 - успешное выполнение операции;
= 1 - цикл вывода звука не активен.

Использование функций драйвера для проигрывания VOC-файлов

Давайте подытожим разговор об используемых функциях. Вообще-то они потрясающе просты и вы наверняка сможете самостоятельно написать небольшую программу и поэкспериментировать с цифровыми каналами ввода-вывода звуковой карты Sound Blaster... Как бы не так! Неужели я вас брошу одних в этом цифровом чистилище? Ни в коем случае. Давайте-ка вместе загрузим драйвер и проиграем какой-нибудь VOC-файл. То, что у нас получится, вы можете впоследствии использовать в собственных играх.

Первое, что мы должны сделать, это загрузить CT-VOICE.DRV в память. Для этого нужно просто выделить некоторое количество памяти, открыть файл как двоичный и загрузить его байт за байтом. Есть, правда, одна проблема: драйвер должен быть загружен от границы сегмента. Это значит, что сегмент может, быть любым, но смещение начала драйвера должно быть нулевым. Ни одна функция из семейства allocate этого делать не умеет. Здесь необходима функция, которая способна резервировать память на границе параграфа. Такая функция называется _dos_allocmem(). Итак, нам надо сделать следующее:

- Открыть файл CT-VOICE.DRV;
- Определить его размер;
- Отвести под него память;
- Загрузить его.

Это делает функция, приведенная в Листинге 9.1.

Листинг 9.1. Выделение памяти для CT-VOICE.DRV.

```
////////////////////////////////////////////////////////////
void Voc_Load_Driver(void)
// загрузить ct-voice.drv
int driver_handle;
unsigned errno,segment,offset,num_para,bytes_read;
// открыть файл драйвера
_dos_open("CT-VOICE.DRV", _O_RDONLY, &driver_handle);
// выделить память
num_para = 1 + (filelength(driver_handle))/16;
_dos_allocmem(num_para, &segment);
//установить указатель на область данных драйвера
_FE_SEG(driver_ptr) = segment;
_FP_OFF(driver_ptr) = 0;
// загрузить код драйвера
data_ptr = driver_ptr;
do {
_dos_read(driver_handle,data_ptr, 0x4000, &bytes_read);
data_ptr += bytes_read;
} while (bytes_read==0x4000);
// закрыть файл
_dos_close(driver_handle);
} // конец функции
```

Мы можем разбить функцию из Листинга 9.1 на три части:

- Вначале мы открываем файл CT-VOICE.DRV в чисто двоичном режиме.Мы не должны делать никаких преобразований символов - это было бы катастрофой! Мы же читаем реальный код, а не ASCII-файл;
- Затем программа вычисляет длину файла и выделяет под него соответствующее количество памяти. Отметим, что мы резервируем память блоками, причем каждый блок — это параграф из 16 байт;
- Наконец, драйвер загружается по 32K за один прием. Эта одно из замечательных отличий функции _dos_read () от стандартной функции getch(): мы можем читать большие

куски кода за один раз.

Теперь, после загрузки драйвера мы должны сохранить указатель так, чтобы впоследствии можно было найти начало драйвера. Этот указатель мы будем хранить в виде глобальной переменной с именем driver_ptr. (Я думаю, что это имя достаточно содержательно.)

Замечание

Давайте немного отвлечемся. Когда вы пишете компьютерные игры (или любое другое программное обеспечение), пожалуйста, используйте такие имена файлов и функций, которые несут смысловую нагрузку и отражают назначение объекта. Постарайтесь избегать таких имен, как t, j, k и им подобных. Используйте имена типа index_1, sprite_alive и так далее. Поверьте моему опыту: когда вы закончите писать компьютерную игру и вернетесь к ней через неделю, вы подумаете: «Не Фон Нейман ли это написал, да здесь сам черт ногу сломит!» Ведь если вы используете иероглифы вместо имен, кто кроме специалиста по иероглифам сможет в .них разобраться? Правильно? Тогда вернемся к нашим баранам.

Верите вы или нет, но загрузка драйвера была самой трудной частью. Посмотрим на код для загрузки VOC-файла. Он выглядит точно так же, как и функция загрузки драйвера. То есть мы должны:

- Открыть файл в бинарном режиме;
- Отвести под него память;
- Загрузить VOC-файл в отведенный буфер. Это делает функция Листинга 9.2.

Листинг 9.2. Загрузка VOC-файла.

```
char far *Voc_Load_Sound(char *filename,
unsigned char *header_length)
{ // загрузка звукового файла с диска в память
// и установка указателя на его начало
char far *temp_ptr;
char far *data_ptr;
unsigned int sum;
int sound_handle,t;
unsigned errno, segment, offset, num_para, bytes_read;
// открыть звуковой файл
_dos_open(filename, _O_RDONLY, &sound_handle);
// выделить память
num_para =1 + (filelength(sound_handle))/16;
_dos allocmem(num_para, &segment) ;
// установить указатель на выделенную память
_FP_SEG(data_ptr) = segment;
_FP_OFF(data_ptr) = 0;
// загрузить звуковые данные
temp_ptr = data_ptr;
do
{
dos_read(sound_handle,temp_ptr, 0x4000, &bytes__read) ;
temp_ptr += bytes_read;
sum+=bytes_read;
} while(bytes_read==0x4000);
// Проверить на всякий случай, звуковые ли это данные.
// Для этого проверяется присутствие слова "Creative".
if ((data_ptr[0] != 'C') || (data_ptr[1] != 'r'))
{
printf("\n%s is riot a voc file!",filename);
_dos_freemem(_FP_SEG(data_ptr) ) ;
return(0);
} // конец звукового файла
header_length = (unsigned char)data_ptr[20];
// закрыть файл
_dosclose(sound_handle) ;
return(data_ptr) ;
} // конец функции
```

Наиболее замечательное в этой функции то, что она возвращает адрес области памяти (точнее, указатель на него), куда был загружен VOC-файл. Мы используем этот указатель позже при проигрывании звука.

Все остальные функции, которые нам потребуются, тривиальны. Мы используем ассемблерные вставки для настройки регистров и команду процессора CALL для вызова драйвера с помощью указателя driver_ptr.

Рассмотрим несколько таких функций, чтобы понять особенности их написания. Драйвер должен быть инициализирован прежде, чем мы сможем его использовать. Это очевидное

требование. Делать это мы должны с помощью функции 3, «Инициализировать драйвер». Листинг 9.3 содержит текст программы этой функции.

Листинг 9.3. Инициализация драйвера.

```
int Voc_Init_Driver(void)
{
// инициализация драйвера функция возвращает слово состояния
int status;
_asm
{
mov bx,3        ; функция инициализации драйвера имеет номер 3
call driver_ptr ; вызов драйвера
mov status,ax   ; сохранение номера версии
} // конец ассемблерной вставки
// возвратить слово состояния
printf("\nDriver Initialized");
return(status);
} // конец функции
```

Другая важная функций сообщает драйверу адрес переменной для передачи слова состояния операции. Текст программы, которая устанавливает эту переменную (я назвал ее ct_voice_status), содержится в Листинге 9.4.

Листинг 9.4. Установка переменной слова состояния драйвера.

```
Voc Set_Status_Addr(char _far *status)
{
unsigned segm, offm;
segm = _FP_SEG(status);
offm = _FP_OFF(status) ;
asm{
mov bx,5        ; функция задания переменной слова состояния
; имеет номер 5
mov es,segm     ; в регистр ES загружается сегмент переменной
mov di,offm   ; в регистр DI загружается смещение переменной
call driver_ptr ; вызов драйвера
} // конец ассемблерной вставки
} // конец функции
```

Наконец, посмотрим реализацию функции 6, «Начать вывод», которая используется для проигрывания звука, хранящегося в памяти. Листинг 9.5 содержит текст этой программы.

Листинг 9.5. Проигрывание VOC-файла из памяти.

```
int Voc_Play_Sound(unsigned char far *addr,
unsigned char header_length)
{
// проигрывает загруженный в память VOC-файл
unsigned segm, offm;
segm = _FP_SEG(addr);
offm = _FP_OFF(addr) + header_length;
_asm{
mov bx,6        ; функция 6 - воспроизведение VOC-файла
mov ax,segm
mov es,ax       ; в регистр ES загружается сегмент
mov di,offm     ; в регистр DI загружается смещение
call driver_ptr; вызов драйвера
} // конец ассемблерной вставки
} // конец функции
```

Функция Voc_Play_Sound из Листинга 9.5 работает следующим образом:

- Адрес VOC-файла, который мы хотим проиграть, из памяти передается в функцию;
- Затем функция использует две крайне полезные макрокоманды FP_SEG () и FP_OFF(), получая таким образом сегмент и смещение стартового адреса буфера с VOC-файлом;
- Сегмент и смещение помещаются в регистровую пару ES:DI в соответствии с требованиями драйвера;
- Вызывается драйвер.

И, пожалуйста - звучит музыка!

В первых ваших собственных играх, так же, как и в игре, которую мы напишем в этой книге, я полагаю, вы будете придерживаться оцифрованных звуков. Воспроизведение такой музыки не слишком сложно, однако и это все же требует некоторых усилий и понимания. В Листинге 9.6

показан полный текст программы, включающий простое меню для воспроизведения нескольких простых мелодий. Перед запуском программы убедитесь, что;

- Все VOC-файлы расположены в текущем каталоге;
- CT-VOICE.DRV также находится в текущем каталоге.

Последнее замечание по поводу воспроизведения оцифрованного звука: после того, как ваша программа начнет проигрывание мелодии, она может делать, что угодно. Лучше всего — продолжить игровой цикл. Операции по воспроизведению звука будут полностью выполняться картой Sound Blaster и аппаратным обеспечением прямого доступа к памяти. Вашей программе не нужно ничего делать, за исключением собственно запуска этого процесса (и затем - его остановки), а это занимает всего несколько микросекунд.

Листинг 9.6. Полная программа воспроизведения звука.

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ////////////////////////////////////
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <bios.h>
#include <fcntl.h>
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ ///////////////////////////////
char   far *driver_ptr;
unsigned version;
char _huge *data_ptr;
unsigned ct_voice_status;
// ФУНКЦИИ ////////////////////////////
void Voc_Get_Version(void)
{
// получить версию драйвера и вывести, ее на экран
_asni
{
mov bx,0        ; функция 0 возвращает номер версии
call driver_ptr ; вызов драйвера
mov version,ax ; сохранить номер версии
} // конец ассемблерной вставки
printf("\nVersion of Driver = %X.0%X",
((version>>8) & 0x00ff), (version&0x00ff));
}
// конец функции ///////////////////////////////////////////////////////////
int Voc_lnit_Driver(void)
{
// инициализация драйвера функция возвращает слово состояния
int status;
_asm
{
mov bx,3        ; функция инициализации драйвера имеет номер 3
call driver_ptr ; вызов драйвера
mov status,ax   ; сохранение номера версии
}// конец ассемблерной вставки
// возвратить слово состояния
printf("\nDriver Initialized");
return(status);
} // конец функции
/////////////////////////////////////////////////////////
int Voc_Terminate_Driver(void)
{
// прекратить работу драйвера
_asm {
mov bx,9        ; функция 9 прекращает работу драйвера
call driver_ptr ; вызов драйвера
} // конец ассемблерной вставки
// освобождение памяти
_dos_freemem(_FP_SEG(driver_ptr));
printf("\nDriver Terminated");
} // конец функции
/////////////////////////////////////////////////////////
void Voc_Set_Port(unsigned port) {
// установить адрес порта ввода/вывода Sound Blaster
```

```
_asm
{
mov bx,l        ; функция 1 устанавливает адрес порта ввода/вывода
mov ax,port     ; поместить адрес порта ввода/вывода в регистр AX
call driver_ptr ; вызов драйвера
} // конец ассемблерной вставки
} // конец функции
//////////////////////////////////////////////////////
void Voc_Set_Speaker(unsigned on)
{
// включить/выключить вывод звука
_asm {
mov bx,4       ; функция 4 включает или выключает вывод звука
mov ax,on       ; поместить флаг включить/выключить в регистр AX
call driver_ptr ; вызов драйвера
} // конец ассемблерной вставки
} // конец функции
//////////////////////////////////////////////////////
int Voc_Play_Sound(unsigned char far *addr,
unsigned char header_lehgth)
{
// проигрывает загруженный в память VOC-файл
unsigned segm,offm;
segm = _FP_SEG(addr);
offm = _FP_OFF(addr) + header_length;
asm
{
mov bx,6        ;Функция 6: воспроизведение"VOC-файла
mov ax,segm
mov es,ax       ; в регистр ES загружается сегмент
mov di,offm     ; и регистр DI загружается смещение
call driver_ptr ; вызов драйвера
} // конец ассемблерной вставки
} // конец функции
//////////////////////////////////////////////////////
int Voc_Stop_Sound(void)
{
// прекращает воспроизведение звука
_asm
{ mov bx,8        ; функция 8 прекращает воспроизведение звука
call driver_ptr ; вызов драйвера
} // конец ассемблерной вставки
} // конец функции
//////////////////////////////////////////////////////
int Voc_Pause_Sound(void)
{
// приостанавливает воспроизведение звука
_asm
{
mov bx,10       ; функция 10 приостанавливает
                ; воспроизведение звука
call driver_ptr ; вызов драйвера
} // конец ассемблерной вставки
} // конец функции ////////////////////////////////////////////////
int Voc_Continue_Sound(void)
{ // продолжает воспроизведение приостановленного звука
asm
{
mov bx,11        ; функция 11 продолжает воспроизведение
                 ; приостановленного звука
call driver_ptr  ; вызов драйвера
} // конец ассемблерной вставки
} // конец функции ////////////////////////////////////////////////
int Voc_Break_Sound(void)
{
// прерывает цикл воспроизведения звука
_asm
```

```
{
mov bx,12       ; функция 12 прерывает цикл воспроизведения звука
call driver_ptr ; вызов драйвера
} // конец ассемблерной вставки
} // конец функции ////////////////////////////////////////////////////////
void Voc_Set_DMA(unsigned dma)
{
// устанавливает номер прерывания прямого доступа к памяти
_asm
{
mov  bx,2        ; функция 2 устанавливает номер прерывания
                 ; прямого доступа к памяти
mov ax,dma       ; поместить в регистр AX
                 ; номер прерывания прямого доступа в память
call driver_ptr  ; вызов драйвера
} // конец ассемблерной вставки
} // конец функции ////////////////////////////////////////////////////////
Voc Set_Status_Addr(char _far *status)
{
unsigned segni,offm;
segm = _FP_SEG(status);
offm = _FP_OFF(status);
// задает переменную слова состояния
asm
{
mov bx,5        ; функция задания переменной слова состояния
                ; имеет номер 5
mov es, segm    ; в регистр ез загружается сегмент переменной
mov di, offm    ; в регистр di загружается смещение переменной
call driver_ptr ; вызов драйвера
} // конец ассемблерной вставки
} // конец функции
//////////////////////////////////////
void Voc_Load_Driver(void) {
// загрузить CT-VOICE.DRV
int driver_handle; unsigned errno,segment_offset,num_para,bytes_read;
// открыть файл драйвера
_dos_open("CT-VOICE.DRV", _O_RDONLY, &driver_handle);
// выделить память
num_para= 1 + (filelength(driver__handle))/16;
_dos_allocmem(num_para, &segment);
// установить указатель на область данных драйвера
_FP_SEG(driver_ptr) = segment;
_FP_OFF(driver_ptr) =0;
// загрузить код драйвера data_ptr = driver_ptr;
do
{
_dos_read(driver_handle,data_ptr, 0x4000, &bytes_read);
data_ptr += bytes_read;
} while (bytes_read==0x4000);
// закрыть файл
_dos_close(driver_handle);
} // конец функции
////////////////////////////////////////////////////////
char far *Voc_Load_Sound(char *filename,
unsigned char *header_length)
{ // загрузка звукового файла с диска в память
//и установка указателя на его начало
char far *temp_ptr;
char far *data_ptr;
unsigned int sum;
int sound handle,t; unsigned errno, segment, offset, num_para, bytes_read;
// Открыть звуковой файл
_dos_open(filename, _O_RDONLY, &sound_handle);
// Выделить память
num_para = 1 + (filelength(sound_handle))/16;
_dos_allocmem(num_para,&segment);
```

```
// Установить указатель на выделенную память
_FP_SEG(data_ptr) = segment;
_FP_OFF(data_ptr} = 0;
// Загрузить звуковые данные
temp_ptr = data_ptr;
do
{
_dog_read(sound_handle,temp_ptr, 0x4000, &bytes_read) ;
temp_ptr += bytes_read;
sum +=bytes_read;
} while (bytes_read==0x4000);
// проверить на всякий случай, звуковые ли это данные;
// для этого проверяется присутствие слова "Creative"
if ((data_ptr[0] !='C') || (data_ptr[1] != 'r'))
{
printf("\n%s is not a voc file!",filename);
_dos_freemem(_FP_SEG(data_ptr));
return(0);
} // конец звукового файла
*header_length = (unsigned char)data_ptr[20];
// закрыть файл
_dos_close(sound_handle) ;
return(data_ptr);
} //конец функции
////////////////////////////////////////////////////////
void Voc_Unload_Sound(char far *sound_ptr) {
// удаление звуковых данных из памяти
_dos_freemem(_FP_SEG(sound_ptr));
) // конец функции ////////////////////////////////////////////////////////
void main(void)
{
char far *sounds[4];
unsigned char lengths[4];
int done=0,sel;
Voc_Load_Driver();
Voc_Init_Driver ();
Voc_Set_Port (0x220);
Voc_Set_DMA(5) ;
Voc_Get_Version();
Voc Set_Status_Addr((char _far *) &ct_voice_status) ;
// загрузка звуковых файлов
sounds[0] = Voc_Load_Sound("beav.voc" , & lengths[0]);
soundsll] = Voc_Load_Sound("ed209.voc", &lengths[1]);
sounds[2] = Voc_Load_Sound{"term.voc",   &lengths[2]);
sounds[3] = Voc_Load_Sound("driver.voc"f &lengths[3]);
Voc_Set_Speaker(1);
// главный цикл событий;.позволяет пользователю
// выбрать звуковой файл.
// Заметьте, что воспроизведение текущего звука можно прервать
while(!done)
{
printf("\n\nSound Demo Menu");
printf("\nl - Beavis");
printf("\n2 - ED 209") ;
printf("\n3 - Terminator");
printf("\n4 - Exit");
printf("\n\nSelect One ? ");
scant("%d",&sel);
switch (sel)
{
case 1:
{
Voc_Stop_Sound();
Voc Play_Sound(sounds[0] , lengths[0]);
} break;
case 2:
{
```

```
Voc_Stop_Sound();
Voc_Play_Sound(sounds[1],  lengths[1]);
} break;
case 3:
{
Voc_Stop_Sound(l ;
Voc_Play_Sound(sounds[2] , lengths[2]);
} break;
case 4:
{
done = 1;
} break;
default:
{
printf("\nFunction %d is not a selection.",sel) ;
} break;
} // конец оператора switch
} // конец цикла while
// закончить работу
Voc_Play_Sound(sounds[3], lengths[3]);
// ожидание окончания звуковой последовательности
// слово состояния имеет значение -1 при воспроизведении звука
// и 0 - в;противном случае.
while(ct_voice_status()) {}
Voc_Set Speaker(0);
// выгрузить звуковые файлы
Voc_Unload_Sound(sounds[0]);
Voc_Unload_Sound(sounds[1]) ;
Voc_Unload_Sound(sounds[2]);
Voc_Unload_Sound(sounds[3]);
Voc_Terminate_Driver ();;
} // конец функции main
```

Частотный синтезатор

Важнейшей частью звуковой карты Sound Blaster является частотный синтезатор. Как я уже говорил в этой главе, чазтотный синтезатор создает нужный сигнал путем модулирования несущей волны. Это позволяет получать гармоники. Более того, с помощью этого метода можно аппроксимировать частотные характеристики звуков реальных музыкальных инструментов и настоящего человеческого голоса.

Частотный синтезатор имеет 18 управляющих блоков, каждый из которых состоит из двух частей:

- Блок модулятора;
- Блок несущей.

Рисунок 9.8 показывает блок типичного частотного синтезатора. Здесь A -общая амплитуда, Wc - частота несущей волны (радиан/с), Wm - частота модулятора (радиан/с).

Выходной сигнал блока модулятора добавляется к сигналу блока несущей волны. В результате этого одна волна «оседлывает» другую. Математически это описывается произведением двух синусоидальных волновых функций. Выходная функция имеет следующий вид:

```
F(t) = A * sin(Wc * t + I * sin(Wm) * t)
```

К сожалению, у нас нет времени надолго задерживаться на разговоре об устройстве синтезатора. Нам надо писать игру! (Хотя, конечно, нам и очень нужна потрясающая музыка!)

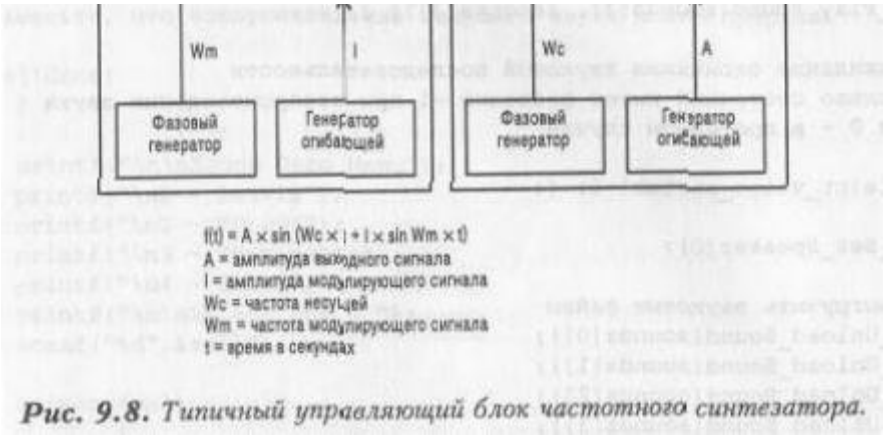


***Рис. 9.8.** Типичный управляющий блок частотного синтезатора.*

Музыка И MIDI

Так же, как и в случае с частотным синтезатором, у нас нет достаточного времени, чтобы разобраться **во** всех тонкостях воспроизведения музыкич с помощью Sound Blaster, Мы лишь затронем этот допрос для того, чтобы у вас сложилось общее представление об этом. Я хотел бы, по крайней, мере рассказать о том, что вам, возможно, пригодится в будущем. Sound Blaster поставляется вместе с драйвером для поддержки воспроизведения MIDI-музыки. (Если вы уже забыли, MIDI - это *интерфейс электромузыкальных инструментов.)* Этот драйвер похож на CT-VOICE.DRV, который мы обсуждали выше, и называется SBFMDRV.EXE. Функции этого драйвера приведены в таблице 9.3.

Табл. 9.3. Функции драйвера FMDRV.DRV для Sound Blaster

Функция	Описание
0	Получить версию драйвера
1	Установить адрес байта состояния
2	Установить таблицу инструментов
3	Установить частоту системного таймера
4	Установить частоту таймера драйвера
5	Транспонировать музыку
6	Начать воспроизведение музыки
7	Остановить воспроизведение музыки
8	Инициализировать драйвер
9	Приостановить воспроизведение музыки
10	Возобновить воспроизведение музыки
11	Задать пользовательское прерывание для системы

Это все. что я хотел сказать о музыкальном драйвере. Я могу поспорить, что мастеру в написании компьютерных игр этой информации вместе с теми программами, что мы написали для

CT-VOICE.DRV, вполне достаточно.

Использование звука в играх

Я, конечно, не могу не рассказать о правильном и своевременном использовании звуков в компьютерных играх. Я видел людей, которые приходили в экстаз от того, что именно в нужный момент в игре звучал гармонирующий с сюжетом звук.

Так как ваши возможности ограничены количеством памяти компьютера, вы должны тщательно подбирать звуки и использовать их действительно в тех местах, где они могут усилить впечатление от созданного вами мира. Все звуки должны быть немного преувеличены и быть чуть длиннее, чем в реальной жизни. Никому не интересно услышать реальный звук винтовочного выстрела, который звучит достаточно тривиально. Игрок хочет слышать даже полет фотонной торпеды! Ну и что из того, что так не бывает? Ведь мы в своем мире полубоги, следовательно, там мы можем делать все, что захотим!

Давайте поговорим об архитектуре звукового интерфейса компьютерных игр. Нам требуется, чтобы разные звуки исполнялись как функция от времени. Для примера возьмем игру, которую мы собираемся сделать, и которую я назвал Warlock. В течение игры, возможно, сложится такая ситуация, которая требует воспроизведения нескольких оцифрованных звуков одновременно. К сожалению, у нас есть только один цифровой канал, и поэтому мы можем исполнять только один поток данных. К счастью, у этой проблемы есть решение. Но мы об этом поговорим немного позже.

Итак, обсудим основные идеи, которых мы должны будем твердо придерживаться в любом случае. В Warlock я хочу использовать рычание, шум ветра и звуки, издаваемые оружием. Вполне может случиться так, что все эти звуки должны будут раздаваться одновременно. Мы должны создать систему приоритетов для звуков и реализовать процедуру вытеснения звуков с низким приоритетом звуками с высоким приоритетом. Один из алгоритмов, который мы можем использовать, приведен ниже.

Алгоритм 9,1. Псевдокод планировщика звуков.

```
while(true)
// оценить время поставленных в очередь звуковых эффектов.
// Если звук уже неактуален, удалить его из очереди. Если нет
// исполняемого звукового эффекта, начать воспроизведение
// первого звука в очереди
else
// если запрашиваемый звуковой эффект имеет более высокий
// приоритет, чем текущий звук (например, игрок выстрелил
// из ружья) - прервать текущий звук и начать исполнение
// эффекта с более высоким приоритетом
else
// если запрашиваемый звуковой эффект имеет меньший приоритет,
// чем текущий звук, установить его в очередь
end
```

Алгоритм 9.1 - весьма хороший черновой пример планировщика. Он позволяет организовать очередь звуков и контролировать их исполнение.

Но все-таки если во время компьютерной игры может раздаваться только один звук в каждый момент времени, это будет слишком скучно. Как же воспроизводить несколько звуков одновременно? Мы можем это сделать с помощью одного трюка.

Представьте, что звук выстрела из лазерного пистолета и звук рычания страшного чудовища должны раздаться одновременно. Предположим также, что у них одинаковый приоритет. По Алгоритму 9.1 мы должны бросить жребий и сыграть их последовательно. Но почему нельзя сложить звуки вместе и получить требуемый результат?

Это вполне возможно. Мы используем теорему о суперпозиции, которая говорит о том, что две любые волны в пространстве могут накладываться одна На другую и в результате порождать новую волну. В нашем случае мы должны сложить данные VOC-файлов.

Есть только проблема в применении этого метода: пространство и время безграничны, но восемь битов остаются восемью битами! Во время сложения двух волн мы можем выйти за пределы восьми или шестнадцати бит. Здесь можно поступить по-разному:

- Мы можем просто отсечь звук по границе восьми или шестнадцати бит;
- Мы можем брать только процентное соотношение волн, что гарантирует попадание их суммы в нужные нам пределы.

Алгоритм 9.2 представляет собой один из вариантов процедуры наложения звуков.

Алгоритм 9.2. Псевдокод наложения звуков.

```
length one=length (sound one)
length two=length (sound two)
if length one > length two
// обработать первую часть звука
for index==0 to length two new sound [index] = .5 * sound onetindex]+ .5 * sound
two[index]
```

```
// обработать оставшуюся часть звука, // просто копируя его с уменьшением for
index=length two+1 to length one
new sound [index] =.5 * sound one[index]
else
// обработать первую часть звука
for index=0 to length one
new sound [index] =.5 * sound one[index]+ .5 * sound two[index]
```

```
// обработать оставшуюся часть звука,
// просто копируя его с уменьшением
for index= length one+1 to length two
new sound [index] =.5 * sound two[index] end
```

По сути, Алгоритм 9.2 — это все, что вам нужно для того, чтобы сложить звуки вместе. Конечно, здесь складываются только два звука, но алгоритм легко может быть приведен к более общему виду. Проблема только в том, что вам потребуется больше времени и памяти. Если у вас в память загружены два VOC-файла по 60K, вы должны будете сделать 60000 сложений, на что уйдет несколько миллисекунд. Это вполне терпимо, но результат сложения нужно поместить в новый буфер размером 60K. Так что, будьте осторожны!

ИТОГ

В этой главе мы поговорили об искусстве озвучивания игр для персонального компьютера;

- Мы изучили Sound Blaster и исполнение на нем оцифрованных звуков, с использованием специального драйвера CT-VOICE.DRV, поставляемого фирмой Creative Labs;
- Мы рассмотрели основы теории звуков и возможности Sound Blaster;
- Мы поговорили о планировщике звуковых эффектов и о ряде других достаточно сложных вещей.

Настало время поговорить о музыке. Встретимся в следующей главе!

РАЗРАБОТКА МУЗЫКИ ДЛЯ КОМПЬЮТЕРНЫХ ИГР

Теперь вы знаете о методах оптимизации и других графических трюках и хитростях. Вы уже научились тому, как сделать графику ослепительной, как наделить игру искусственным разумом и, может быть, даже умеете считывать данные из портаджойстика. В этой главе мы расскажем о самом мощном методе длительного эмоционального воздействия на игрока. Звук и музыка позволят вам непосредственно управлять эмоциями пользователя в соответствии с контекстом вашей игры. Хорошо разработанные звуковые эффекты могут создать даже более яркую «виртуальную реальность», чем самая изощренная трехмерная графика. Эмоциональное воздействие музыки очень разнообразно, она может передавать ощущение страха, волнения, победы, грусти и даже ужаса.

Музыкальные и звуковые эффекты - это наиболее мощный инструмент, способный воздействовать на игрока на уровне эмоций. Если для вас это не очевидно, посмотрите снова фильм Спилберга «Парк юрского периода», В тот момент, когда тиранозавр появляется, чтобы съесть Ленд Ровера, выключите звук и тогда вы поймете, что весь ужас ситуации полностью создается за счет музыки и звука. Если графика позволяет игроку увидеть вашу «виртуальную реальность», то музыка и звуковые эффекты дают ему возможность почувствовать ее.

Хотя для многих из нас все это интуитивно понятно, вы должны практически проверить это на себе. Возьмите, например, «Терминатор II», «Конан-Варвар», «Звездные войны», «Чужие» или «Парк юрского периода». Каждый раз, когда по ходу фильма ваше сердце учащенно забьется, закройте глаза и попробуйте проанализировать только звуковое сопровождение, которое вы сейчас слышите. Послушайте, как тесно переплетается основная мелодия музыкальной партитуры фильма с его отдельными эпизодами, позволяя им зарождаться и развиваться. Как-нибудь, во время действительно шумной сцены, выключите звук. Вы почувствуете, что напряженность ситуации исчезла, словно в темной комнате вдруг зажгли свет.

Нет ничего из того, что создается в фильмах звуком и музыкой, что нельзя было бы повторить с помощью мультимедиа. За одним только исключением. Наши звуковые эффекты и музыка должны быть интерактивны и должны совпадать с контекстом игры. Добавление интерактивных элементов в звуковое сопровождение игры многократно увеличивает ее эмоциональную напряженность.

Звуковое сопровождение содержит четыре основных компонента: диалоги, звуковые эффекты, фон и музыку. Ниже мы коротко рассмотрим каждый из этих компонентов и их роль в компьютерной игре.

Диалоги. Большинство диалогов, которые вы слышите в кино, переделываются в студии уже после того, как сцена была отснята. Эта процедура позволяет актеру (или актрисе) сфокусироваться на том, как звучат его (или ее) слова и, кроме того, позволяет звукорежиссеру контролировать точный аудиобаланс в окончательной звуковой дорожке. Все ваши диалоги должны быть сделаны профессионально и, желательно, в студии звукозаписи, Однако если у вас нет продюсера с громадным капиталом, вы, вероятно, не сможете ангажировать актеров Голливуда и снять классную студию звукозаписи. Однако вы можете привлечь к своей работе одного из специалистов в области мультимедиа. Обычно такие специалисты предлагают полный спектр услуг по озвучиванию компьютерных программ, включая наем актеров, звуковые эффекты на заказ и микширование звуковой дорожки. Кроме того, они могут представить звуковые данные в том формате, который требуется для вашего программного и аппаратного обеспечения.

Звуковые эффекты. Несмотря на доступность все большего количества звуковых библиотек, во многих случаях все же требуется создание своих оригинальных звуковых эффектов. Естественно, если что-то из библиотек подходит для вашей игры, вы можете это использовать, но в любом случае стоит проконсультироваться с опытным,в области мультимедиа специалистом, чтобы создать для своего проекта ряд оригинальных звуковых эффектов.

Фон. Фон - это те шумовые эффекты, на которые вы при просмотре фильма не обращаете внимания, однако их отсутствие ощущается мгновенно. Фон включает в себя звуки шагов, шумы автомобилей, ветра, щебетание птиц и другие звуки, окружающие нас в повседневной жизни. Само собой разумеется, что фон должен быть связан с контекстом фильма. В кино каждый шаг и фабричный гудок добавляется в звуковую дорожку и покадрово синхронизируется уже после того, как фильм снят.

Шумовые эффекты создают большее ощущение «виртуальной реальности», чем точнейшая компьютерная графика. Воздействие шумовых эффектов многократно усиливается, если они проходят специальную цифровую обработку типа Qsound или реверберации. Реверберация — это метод обработки звукового сигнала, в результате которого на исходный звук накладывается эхо и отражение от поверхностей, так же, как это происходит в реальной обстановке. Например, для звуковой карты Creative Labs AWE 32 можно запрограммировать точные геометрические характеристики комнаты для каждого момента времени. Это делается с помощью механизма событий MIDI, при этом все шумовые эффекты звучат так, как если бы они происходили в настоящей комнате определенной формы и размеров.

Фон и цифровые эффекты - самая интерактивная составляющая звуковой дорожки. Благодаря шумовым эффектам игрок слышит звук шагов, грохот выстрелов, рычание приближающихся монстров, завывание ветра, щебет птиц, шум улицы. Воспроизведение таких

звуков в реальном времени и в полном соответствии с вашей «виртуальной реальностью», позволяет игроку глубоко погрузиться в созданный вами мир. Удачное фоновое оформление усиливает ощущение «виртуальной реальности» больше, чем всевозможные очки, перчатки, шлемы и другие приспособления для компьютерных игр. Пренебрегая возможностями звуковых эффектов, вы отбрасываете пользователя во времена немого кино.

Музыка. Уже много было сказано по поводу того, как адаптировать музыку к интерактивной природе продуктов мультимедиа. В фильме события происходят последовательно. Поэтому композитор точно знает, с какого момента необходимо начинать нагнетать страсти. В компьютерных играх этот момент неизвестен; он зависит от того, когда же, наконец, игрок откроет ту самую «дверь, ведущую в Ад». Поэтому в одних играх просто в разных эпизодах звучат разные мелодии, тесно не связанные с происходящим, в других же авторы пробуют экспериментировать с разветвлением MIDI-музыки, пытаясь создать плавные переходы от одного сюжета к другому. Были даже попытки создавать алгоритмическую музыку, то есть музыку, синтезируемую компьютером в реальном времени.

Вероятно, наиболее оптимальным решением будет собрать вместе всевозможные эмоции, возникающие по ходу игры, и обратиться к профессиональному композитору. То есть поступить так, как если бы вы снимали фильм. Композитор создаст музыкальный фон для каждой из ситуаций, а вы сможете его использовать по контексту игры. Таким образом, с одной стороны, эти мелодии не будут написаны для четко определенных моментов игры, а с другой, они все же будут связаны с эмоциональным контекстом. Поэтому, когда во время игры пользователь окажется в нервозной ситуации или попадет в опасность, вы с помощью музыки сможете усилить эти ощущения.

Другой подход к музыкальному сопровождению заключается в использо вании музыки просто для передачи общего настроения текущего эпизода. Основной же упор в этом случае делается на фон, диалоги и звуковые эффекты. Несомненно, звуки выстрелов, взрывов или вопли ужаса -отличный способ доходчиво рассказать игроку о том, что же происходит в игре в данный момент.

Нам, как разработчикам игр, очень повезло, что мы имеем в своем распоряжении очень мощные инструменты для создания звукового оформления. А ведь еще пару лет назад вы не смогли бы добиться от компьютера ничего, кроме бибиканья, свиста и треска. Естественно, что такое звуковое сопровождение вызывало желание побыстрее отыскать клавишу «выключить звук». Возможности первого поколения звуковых карт не многим отличались от возможностей встроенного динамика персонального компьютера. Несмотря на то, что звуковые карты, типа Adiib Personal Music System позволяли добавить в игру Интерактивные звуковые эффекты, их эмоциональное воздействие было еще очень ограниченным. Основная слабость пресловутых частотных синтезаторов состояла в том, что эмоциональный накал их музыки мог соревноваться разве что с мелодией, исполненной на детской дудочке.

К счастью, положение вещей кардинально улучшилось с появлением дисков CD-ROM, цифровых звуковых карт и MIDI-синтезаторов с волновыми таблицами. Теперь компьютерные звуки и музыка могут поспорить в эмоциональном наполнении даже с картинами Спилберга. Я не оговорился, действительно могут! Ведь когда мы смотрим картины Спилберга, мы не можем принимать непосредственного участия в событиях. Мы видим, как динозавр атакует Ленд Ровера, но мы не можем вмешаться в ситуацию. В компьютерной же игре мы можем попытаться спастись от динозавра. Причем, когда мы будем убегать от злобного зверя, музыка и звуковые эффекты тесно сплетутся с нашими действиями. В результате наши нервы будут напряжены до предела — и не стоит забывать, что такого единства действия и звука можно достичь только в интерактивном мире!

Игра DOOM фирмы Id Software - один из лучших примеров использования интерактивного цифрового звука; Кто из вас не сжимался от страха на своем стуле, слыша, как из какого-нибудь угла раздается жуткое рычание и храпение монстра. Обратите внимание на то, что вы так бурно реагируете, даже еще не видя, а только слыша чудовище. Эмоциональное напряжение так велико, что когда, наконец, монстр выскакивает из-за укрытия, а затем, зашатавшись, падает под градом ваших пуль, вы испытываете самое настоящее чувство огромного облегчения. Все это достигается в немалой степени именно за счет звуковых эффектов, создающих у игрока соответствующее настроение. Удачный эффект помогает пользователю еще глубже окунуться в мир созданной вами игры.

Но я должен вас предостеречь. Насколько правильное использование звуковых эффектов и музыки усиливает игру, настолько же их плохое качество портит ее. Неудачные и не соответствующие эмоциональному настрою игры эффекты и музыка - это потеря денег, времени и дискового пространства. Даже если плохой, непрофессионально сделанный звук и не погубит окончательно вашу игру, он, несомненно, произведет на игрока отрицательное впечатление независимо от качества всей остальной игры.

Для того чтобы избежать подобных ошибок и сделать музыку и звук в вашей игре максимально эффективными, поступайте следующим образом:

- Используйте звуковые эффекты, созданные профессионалами. Для этого либо обращайтесь к специалисту по мультимедиа, либо очень тщательно отбирайте эффекты из звуковых библиотек. Не воруйте звуковые эффекты из фильмов, с пластинок и телевидения. Все они защищены авторскими правами. Распространение вашей игры в этом случае будет запрещено и вам даже возможно придется объясняться в суде. Если вы считали замечательные звуковые эффекты из «Звездных войн» с BBS — это еще не значит, что вы имеете право их использовать в игре. Если в библиотеках звуковых клипов вам не удается найти звуковые эффекты, полностью удовлетворяющие вашим требованиям, остается только обратиться к специалисту по мультимедиа;

- Используйте музыку, написанную профессионалами. Для этого либо обратитесь к композитору, создающему компьютерную музыку, либо используйте подходящие к вашей игре высококачественные клипы. Учтите, что композитор должен работать именно в жанре компьютерной музыки, иначе у него возникнут такие же трудности, как у бас-гитариста, который вдруг решил играть на саксофоне. Поэтому то, что данный человек — великий музыкант, еще не значит, что он способен создать классную midi-композицию для вашей игры. Необходимо, чтобы композитор умел обходить все слабые стороны и эффективно использовать все преимущества MIDI;
- Убедитесь, что в любой момент времени музыка поддерживает эмоциональное напряжение в игре и не расходится с контекстом. Помните об интерактивной природе вашей музыки и о том, что она должна быть связана с контекстом игры. Чем больше внимание вы уделите этому, тем сильнее будет эффект;
- Посмотрите фильмы, похожие на вашу игру. Каждый раз, когда вы будете их смотреть, попробуйте осознать, каким образом ваши эмоции усиливаются звуковыми эффектами и музыкой. Если вы не обладаете хорошим музыкальным слухом или если вы четко не представляете, как эффективно использовать звук и музыку в игре, вам следует передать эту часть проекта кому-нибудь другому;
- Прислушивайтесь к мнению своего композитора. Найдите композитора с опытом работы с вашим аппаратным обеспечением и хорошо ориентирующегося в мультимедиа. Очень четко объясните композитору, что вам нужно. Дайте ему образцы музыки с CD или из фильмов, эмоционально совпадающих с вашей игрой. Ваше сотрудничество будет плодотворнее, а музыка лучше, если вы ясно изложите композитору, чего вы с помощью его музыки хотите достичь в своей игре. Если же композитор не поймет вашего замысла, то вы только понапрасну потратите деньги, нервы и время.

Именно с помощью эффективного использования звуковых эффектов и музыки преодолевается тот самый психологический барьер, который мешает пользователю забыть, что он «всего лишь играет» и полностью погрузиться в созданный вами мир. Я думаю, что вы уже на собственном опыте убедились, насколько важны для восприятия игры удачное сочетание звуковых эффектов и музыки.

Способы воспроизведения звука

Рассмотрим способы, с помощью которых вы можете воспроизводить звук на персональных компьютерах.

Цифровой звук. Выпуск фирмой Creative Labs платы Sound Blaster заложил фундамент для использования оцифрованных звуков персональными компьютерами. Все последующие звуковые карты (такие как Covox Speech Thing, The Walt Disney Sound Source или Media Vision ProAudio Spectrum) поддерживали эту возможность. Цифровой звук позволяет вашей программе исполнять все, что может быть записано с помощью микрофона, включая звуковые эффекты, человеческую речь и музыку. Конечно, цифровой звук требует огромных объемов памяти и дискового пространства, но воспроизведения в играх звуковых эффектов и диалогов очень эффективно.

Цифровой звук - это как раз то, что вы слышите при воспроизведении музыкальных записей с компакт-дисков. Звук представляет собой волну определенной формы. Звук, воспроизводимый динамиком, — это, фактически, напряжение, подаваемое на катушку динамика и заставляющее мембрану двигаться вперед и назад в соответствии с силой сигнала. Движения мембраны динамика генерируют в воздухе звуковую волну. И наоборот, звуковая волна может быть переведена в цифровую форму, пугем преобразования уровня напряжения в числовые значения. Этот процесс называется аналого-цифровым преобразованием или АЦП. Форма звуковой волны приближенно описывается некоторым множеством повторяющихся выборок. Таким образом, звуковая волна, преобразованная в цифровую форму, имеет две важные характеристики: число использованных битов для представления данных и частоту, с которой звуковая волна была оцифрована. На стандартный звуковой компакт-диск музыка записывается с разрешением 16 бит и частотой 44КГц в стереозвучании. Это значит, что каждая выборка имеет 32 бита и выборки производятся 44 тысячи раз в секунду. Таким образом, одна секунда звука на компакт-диске требует 176000 байт. Одна минута - уже свыше 10 мегабайт!

Обычно компьютерные игры используют звуковые данные со значительно более низким разрешением, так как проигрывание цифрового звука с качеством компакт-диска потребует всех ресурсов машины. Как правило, игры имеют дело с 8-битными выборками на частоте 11 КГц. Но даже на такой частоте совсем короткий звук легко займет всю память машины. Так что, хотя цифровой звук и является лучшим способом создания разнообразных звуковых эффектов, таких как выстрелы и взрывы, использовать его для музыкального оформления компьютерных игр чересчур накладно.

Другая проблема заключается в том, что, хотя оцифровка звука и позволяет проигрывать музыкальные записи, с ее помощью практически невозможно интерактивно управлять музыкальным фоном в соответствии с текущим контекстом игры. Для устройств MIDI такой проблемы не возникает, так как воспроизведение осуществляется под полным контролем со стороны программного обеспечения. Другими словами, вы можете в реальном времени посылать компьютеру эквивалент нотной записи.

Частотные синтезаторы. Самой первой популярной звуковой картой для персонального компьютера была плата Adiib Personal Music System. Она содержала синтезатор частотной модуляции Yamaha YM3812 (OPL2) FM. Э".э квазипрограммируемое устройство способно с помощью осцилляторов формировать вид звуковой волны, используя частотную модуляцию, генератор белого шума и операторы нарастания, удержания, затухания и освобождения звука. Это

звучит замысловато не только на словах, но и на деле. Устройство феноменально сложно программируется. Даже приложив всю вашу изобретательность при программировании частотного синтезатора, вам вряд ли удастся достичь приемлемых результатов. К счастью, с появлением нового поколения MIDI-синтезаторов с волновыми таблицами прямое программирование частотных синтезаторов уходит в прошлое. Существует „много .систем, позволяющих YM3812 эмулировать устройство MIDI, избавляя программистов от необходимости работать с замысловатой логикой этого устройства.

MIDI. Спецификация интерфейса электромузыкальных инструментов (MIDI) - это международный стандарт «де-факто», который определяет последовательный интерфейс для объединения музыкальных синтезаторов, музыкальных инструментов и компьютеров. MIDI поддерживается Ассоциацией производителей MIDI (Лос-Анджелес, Калифорния). Он опирается на аппаратное (каналы ввода/вывода и соединительные кабели) и программное обеспечение (тон, громкость и так далее). В соответствии со спецификацией, принимающее устройство в MIDI-системе должно интерпретировать музыкальные данные даже в том случае, если у передающего устройства нет возможности узнать, что способно делать принимающее устройство. Это чревато проблемами, если приемник не способен корректно интерпретировать данные. В обобщенном MIDI эта проблема решается предварительной идентификацией возможностей аппаратуры.

Все устройства обобщенного MIDI поддерживают звучание музыкальных и ударных инструментов, вместе со 128 звуковыми эффектами. Системы обобщенного MIDI поддерживают одновременное использование 16 MIDI-каналов с минимум 24 нотами в каждом и имеют определенный набор музыкальных контроллеров. Это означает, что в случае обобщенного MIDI передающее устройство знает, что ожидать от принимающего. Точно так же и файл, созданный одним устройством, соответствующим спецификациям обобщенного MIDI, распознается любым другим подобным устройством без потери нот или нарушения баланса инструментов при воспроизведении. Это, однако, справедливо только в теории. На практике же файл MIDI обычно имеет характеристики, существенно зависящие от конкретного устройства. Это обусловлено тем, что если сами инструменты и стандартизованы в обобщенном MIDI, то их баланс, тембр и качество — нет. Таким образом, каждый обобщенный MIDI-инструмент играет с разными характеристиками. При написании произведения MIDI для качественного исполнения на любом устройстве композитору не следует использовать специфические особенности конкретного синтезатора. Так как файлы MIDI обычно небольшие, то я, чтобы полностью использовать все возможности каждого синтезатора, обычно делаю различные версии для наиболее популярных MIDI-синтезаторов.

Большинство поставщиков придерживаются обобщенного стандарта MIDI, в том числе Roland (Roland Sound Canvas, Roland RAP-10), Creative Labs (AWE 32), Logitech- (Logitech SoundWave), Ensoniq (SoimdScape), Gravis (Gravis Ultrasound), Turtle (Turtle Beach Maui), Sierra Semiconductor (Aria). Кроме того, устройства с частотными синтезаторами, такие как Sound Blaster, также могут эмулировать устройства MIDI с помощью набора драйверов типа MIDPAK. Видимо, в будущем вся интерактивная музыка будет придерживаться стандарта .MIDI. Благодаря этому вы сможете качественно воспроизводить с помощью разнообразных звуковых карт полнооркестровые произведения, созданные для вас профессиональными композиторами. Файлы данных MIDI небольшие, требуют сравнительно небольшого числа прерываний для обслуживания и не сильно загружают процессор. Большинство современных MIDI-синтезаторов используют алгоритмы. Алгоритмы — это оцифрованные записи реальной игры на разнообразных инструментах. Алгоритмы хранятся в постоянном запоминающем устройстве синтезатора. И когда MIDI-файл описывает, например, звучание пианино, то используется оцифрованный звук настоящего пианино. Благодаря этому мы получаем удивительно реалистично звучащую музыку. Полный набор алгоритмов, включающий в себя хоровые эффекты и эффекты реверберации, придаст вашей музыке дополнительный колорит.

MOD. Файлы MOD стали популярны благодаря компьютерам Amiga фирмы Commodore. Феноменальные для своего времени возможности этого игрового компьютера просто поражали. Аппаратное обеспечение этой машины поддерживало четыре канала цифрового звука. Многие создатели игр, сотрудничавшие с фирмой Amiga, хотели, чтобы в их играх звучала музыка. Поэтому они придумали схему, позволяющую получать полнооркестровую музыку, а файлы, содержащие звуковые данные, получили наименование MOD. Практически, этот метод базируется на программной эмуляции частотного синтеза с использованием алгоритмов (о них мы рассказывали выше в разделе MIDI). Надеюсь, вы еще помните, что алгоритмы содержат оцифрованную запись игры на разнообразных инструментах и хранятся в ПЗУ. Когда требуется воспроизвести звук пианино, то модулируется частота настоящей, исполненной на фортепиано, оцифрованной ноты. При использовании технологии MOD-файлов, делается все то же самое, только программным путем. Это требует достаточного количества оперативной памяти для хранения оцифрованных звуковых эффектов и программного обеспечения для частотной модуляции в реальном времени. Такая задача отнимает много памяти и ресурсов процессора. Кроме того, написание музыки в виде MOD-файлов значительно более трудоемкая задача, чем создание композиций для MIDI.

Так что, хотя файлы MOD и являются наиболее полным, наиболее реалистичным методом создания музыки для простых устройств типа Sound Blaster, они все же не могут соревноваться с настоящим синтезатором MIDI. Кроме того, когда цифровой канал звуковой платы занят исполнением музыки, усложняется воспроизведение шумовых эффектов. Так как современный рынок движется в сторону MIDI-синтеза, наиболее разумным будет писать музыку в формате MIDI, а фоновые и шумовые эффекты воспроизводить с помощью оцифрованных звуков. Пакеты программ MIDPAK и DIGPAK как раз и предназначены для этих целей.

Красная книга компакт-дисков. Одно из преимуществ дисководов CD-ROM, которые становятся все популярнее, состоит в том, что они могут проигрывать стандартные компакт-диски. Однако вы не можете сделать так, чтобы ваша игра одновременно и работала с диска CD-ROM и

воспроизводила музыку оттуда же. Доступ к записанным на CD-ROM данным и к звуковым дорожкам является полностью взаимоисключающим. Также невозможно добиться плавной работы программы при постоянном переходе от данных к звуку и обратно. Однако многие разработчики любят помещать музыкальные фрагменты из игр на звуковые дорожки CD-ROM, и вы также можете взять это себе на вооружение.

Программное обеспечение цифрового микшироваиия. За исключением Gravis Ultrasound и Creative Labs AWE32, почти все звуковые карты поддерживают только один канал для цифрового звука. Однако в интерактивной игре вам может потребоваться воспроизвести одновременно несколько звуков. Этого вы можете добиться, применяя программное обеспечение, которое осуществляет цифровое микширование. Так как звук аддитинен, этот процесс удивительно прост. Все необходимые звуки, которые должны звучать в данный момент, складываются в буфере, отсекаются точки, вышедшие за установленные пределы, а затем результат передается звуковой карте. Некоторые инструментальные пакеты для разработки программного обеспечения включают поддержку программного цифрового микширования в свои программные интерфейсы.

Загружаемые алгоритмы инструментов и звуковых эффектов. При использовании звуковых карт Gravis Ultrasound и Creative Labs AWE32 программа может загружать музыкальные инструменты или эффекты в память самой платы. После этого вы можете манипулировать ими, простоинициируя события MIDI. Это очень мощный метод, потому что он позволяет не только осуществлять поддержку многоканальности и использовать оригинальные инструменты и эффекты при низкой загрузке процессора и экономном расходовании памяти. Данный метод еще дает возможность манипулировать звуковыми эффектами в реальном времени, применяя сдвиг высоты звука, управление ударными инструментами и даже хоровые эффекты и эффекты реверберации.

Пакеты программ DIGPAK и MIDPAK

На подступах к новому тысячелетию разработчикам уже не надо беспокоиться о программировании на аппаратном уровне индивидуальных звуковых устройств. В Windows 3.1 уже существует механизм драйверов, позволяющий прикладным программам воспроизводить оцифрованный звук и музыку MIDI. Точно так же в реальном или защищенном режиме процессора существует ряд программных интерфейсов, освобождающих вас от этого тяжкого бремени, позволяя сфокусировать свои силы на написании самих звуков и музыки. Среди подобных коммерческих систем можно назвать Audio Interface Library фирмы Miles Design и Sound Operating System фирмы Human Machine Interfaces.

Одной из наиболее популярных среди разработчиков систем являются созданные фирмой Audio Solution пакеты программ DIGPAK и MIDPAK. Первый из них представляет собой универсальный DOS-интерфейс для исполнения цифрового звука на практически любой звуковой карте. MIDPAK позволяет вам воспроизводить полнооркестровую MIDI-музыку практически на любой звуковой карте, включая прекрасную эмуляцию MIDI для карт, не обладающих подобной возможностью (например, Sound Blaster). С 1 января 1994 года пакеты программ DIGPAK и MIDPAK бесплатны для некоммерческого использования. Небольшая лицензионная плата требуется для коммерческого распространения драйверов. Эта плата идет на усовершенствование драйверов и на содержание BBS поддержки — SoundBytes OnLine BBS.

Пакеты программ DIGPAK и MIDPAK были созданы для того, чтобы разработчик игр для DOS мог справиться с громадным количеством наводнивших рынок звуковых карт. Уже больше пяти лет мы с Джоном Майлсом из Miles Design (как и многие другие специалисты из фирм по производству звуковых плат) занимаемся разработкой, улучшением и обновлением этих драйверов. Пакеты программ DIGPAK и MIDPAK вы найдете на дискете, прилагаемой к этой книге, а чуть ниже в данной главе будет приведена документация на API этих пакетов. Обратите внимание, что на дискете есть дополнительная документация и многочисленные примеры использования драйверов.

Я хотел бы сказать спасибо всем, кто помогал создавать эти драйверы:

- Джон Майлс, Miles Design
- Скотт Синдров, Creativ Labs
- Дуг Коди, MediaVision
- Майк Лейбов, Forte
- Майк Дабс, Simutronics
- Керщен Хеллер, Sierra Semiconductor
- Мило Стрит, Street Electronics
- Брэд Крэг, Advanced Gravis
- Ричард Мазерес, Turtle Beach

Что такое «расширенный MIDI-формат»?

Так как пакет программ MIDPAK использует драйверы AIL Miles Design, то он не проигрывает MIDI-файлы напрямую. Ваш MIDI-файл должен быть преобразован в *расширенный MIDI-формат.* Этот формат был разработан Miles Design. Он поддерживает несколько MIDI-последовательностей в одном файле, причем с помощью команды PlaySequence пакет программ MIDPAK позволяет переходить от одной последовательности к другой практически мгновенно. Процесс добавления нескольких MIDI-файлов в один расширенный MIDI-файл (XMI) выглядит следующим образом. Допустим, у вас есть три музыкальных файла SONGA.MID, SONGB.MID и SONGC.MID, и вы хотите поместить их в один файл XMI. Для этого в командной строке DOS надо выполнить следующую команду:

```
MIDIFORM SOMG.XMI SONGA.MID SONGB.MID SONGC.MID
```

Команда поместит эти три MIDI-последовательности в один файл расширенного MIDI-формата SONG.XMI. Вы можете манипулировать ими из MIDPAK, используя функции PlaySequence(0), PlaySequence(l) и PlaySequence(2).

В каком формате должны быть представлены данные MIDI?

В случае настоящего MIDI-устройства, такого как MPU401 или SoundCanvas, ему будут передаваться данные MIDI без каких-либо изменений (исключая сообщения SysEx).

В случае Adiib или Sound Blaster, MIDPAK будет эмулировать устройство MIDI и эта эмуляция имеет определенные ограничения;

- Каналы 2-9 используются для мелодических инструментов;
- Канал 10 используется для ударных инструментов;
- Загружаемые алгоритмы должны быть в формате обобщенного MIDI.

Вопросы и ответы

В этом разделе приведены ответы на наиболее часто задаваемые вопросы, касающиеся озвучивания игр.

При исполнении MIDI-музыки с помощью пакета программ MIDPAK мне кажется, что часть музыкальной композиции теряется

Проверьте назначение каналов. Эмуляция MIDI на Sound Blaster и других картах происходит на каналах 2-9 для мелодических инструментов и на 10 канале для ударных. Многие программы записывают последовательности MIDI, начиная с канала 1. При эмуляции MIDI пакетом MIDPAK канал 1 игнорируется. Назначения каналов MIDPAK были разработаны для эмуляции Roland MT-32. Несмотря на ряд усовершенствований для поддержки обобщенного MIDI, каналы все еще ограничены номерами со второго по девятый и десятым.

При исполнении MIDI-музыки с помощью пакета программ MIDPAK, она звучит иначе, чем при использовании моего собственного музыкального контроллера. Почему?

Ваш контроллер использует другие алгоритмы, нежели MIDPAK. Чтобы избежать этого, необходимо обратиться к профессиональному композитору, который сможет скорректировать вашу композицию для исполнения с помощью MIDPAK. Однако эти услуги вряд ли будут бесплатными.

Мне не удается изменить громкость с помощью MIDPAK. Почему?

Уровень громкости в MIDPAK изменяется относительно базового уровня данного канала. MIDPAK не может изменить громкость, если вы не определили базовый уровень для каждого канала в вашем файле MIDI. Вы можете определить базовую громкость для MIDI-канала, используя Контроллер 7.

Могу ли я использовать один и тот же MIDI-файл для всех звуковых карт?

Нет, но вы можете сделать нечто похожее. Во-первых, сделайте запись в стандарте обобщенного MIDI. Затем ее нужно скорректировать для Sound Canvas, MIDI, OPL2/OPL3 и MT-32. Базовые уровни громкости и качество звучания разных алгоритмов для этих устройств немного различаются. Исходные тексты программы SETM (программа конфигурации MIDPAK) входят в поставку (файл SETUP.ZIP) и, в зависимости от звукового драйвера выбранного пользователем, вы можете копировать различные версии вашей музыки.

API пакета программ DIGPAK

Пакет программ DIGPAK наряду с интерфейсами реального и защищенного режима DOS включает в себя набор исходных текстов, которые работают во всех моделях памяти. Драйверы DIGPAK полностью совместимы с драйверами MIDPAK. Кроме того, хотя все драйверы DIGPAK разрабатывались для реального режима, они работают в защищенном режиме через подключаемый интерфейс DIGPLAY. Поддержка защищенного режима требует наличия драйверов DIGPAK версии 3.4 или выше.

В версии 3.4 пока имеются не всё драйверы звука DIGPAK. Еще не конвертированы драйверы для Gravis Ultrasound и Turtle Beach Multisound. Кроме того, драйверы, использующие таймер 8253, также не будутработать в

защищенном режиме из-за большой потери производительности и возможных конфликтов вследствие высокой частоты прерываний таймера.

Набор драйверов оцифрованного звука использует вектор прерывания 66h, что обеспечивает прозрачный программный интерфейс. Ниже описываются два способа воспроизведения оцифрованного звука. Первый способ основан на прерываниях. Второй — на использовании библиотеки функций на Си или . ассемблере, которые позволяют не только осуществлять доступ к драйверам звука, но и предоставляют другие полезные функции. Эти функции находятся в исходном файле DIGPLAY.ASM.

```
**************************************************************
********* Спецификация интерфейса прерывания 66h *****************
**************************************************************
```

Вызывая функции для исполнения звука, вы передаете ей адрес структуры звуковых данных (SNDSTRUC), содержащей базовую информацию, описывающую желаемый звуковой эффект. Ниже показано, как это сделать.

```
*** РЕАЛЬНЫЙ РЕЖИМ:
typedef struct
(
unsigned char far *sound* // дальний указатель на звуковые данные
unsigned short sndlen;    // длина звуковой последовательности
short far *IsPiaying;     // адрес флага состояния
short frequency;          // частота воспроизведения
} SNDSTRUC;
*** ЗАЩИЩЕННЫЙ РЕЖИМ:
typedef struct
{
unsigned char *sound* // должен быть в формате
// СМЕШЕНИЕ:СЕГМЕНТ в пределах 1Мб!
unsigned short sndlen;// длина звуковой последовательности < 64K
short *IsPlaying;     // адрес флага состояния. СМЕЩЕНИЕ:СЕГМЕНТ!
short frequency;      // частота воспроизведения
} SNDSTRUC;
********* функции DIGPAK ***************************************
```

Замечание

Все неуказанные номера функций являются устаревшими.

Функция № 1: DigPlay

Функция DigPlay исполняет 8-битный оцифрованный звук. Она является базовой функцией DIGPAK для исполнения звуковых эффектов. Прикладная программа создает структуру звуковых данных, описывая положение звука в памяти и его продолжительность. Функция DigPlay может модифицировать содержимое звукового буфера. Поэтому если вам необходимо исполнить звуковой эффект более одного раза, используйте комбинацию функций MassageAudio и DigPlay2. Вы также можете использовать DigPlay для воспроизведения звука с разным разрешением с компакт-дисков и даже для его записи. Универсальность данной функции объясняется тем, что интерфейс пакета программ DIGPACK развивался в течение нескольких лет и изначально содержал только одну эту функцию. Для обеспечения совместимости сверху вниз, эта функция сохранила прежние возможности, получив расширенные установки различных режимов воспроизведения и записи.

Прототип этой функции находится в файле DIGPLAY.H. Реализация функции содержится в файле DIGPLAY-OBJ, объектном файле, созданном из файла DIGPLAY.ASM. Вы можете использовать либо интерфейсные функции, либо напрямую вызывать прерывание 66h.

Вход: AX=688h Номер команды

DS:SI *Реальный режим*

Указатель на звуковую структуру, описывающую звуковой эффект.

ESI *Прямая адресация*

В случае использования 32-разрядной адресации, ESI указывает на 32-разрядный адрес звуковой структуры в первом мегабайте адресного пространства. Будет преобразован драйвером DIGPAK в допустимый адрес СЕГМЕНТ:СМЕЩЕНИЕ.

Выход: Ничего

Функция № 2: SoundStatus

Данная функция проверяет текущее состояние звукового драйвера и возвращает номер версии для драйверов версии 3.1 и выше. Она используется для запроса текущего состояния исполнения звукового эффекта. Если в настоящий момент звуковой эффект воспроизводится, в регистре AX будет возвращена 1. Данная функция также используется для определения версии звукового драйвера DIGPAK, числовое значение которой помещается в регистр BX. Это справедливо Для драйверов версий 3.1 и выше. Для получения этого значения вы также Можете использовать функцию ReportVersionNumber.

Вход: AX =689h

Выход: AX =0 Звук не исполняется.

=1 В настоящий момент исполняется звуковой эффект.

BX= Начиная с версии 3.1 вызов функции SoundStatus возвращает в регистре BX номер версии. Номер версии возвращается в десятичном формате и умножен на 100, то есть значение 310 соответствует версии 3.10. Версии ниже 3.1 ничего невозвращают в регистре BX, так что вы должны обнулить его перед проверкой номера версии. Если BX остался нулевым, значит версия загруженного драйвера DIGPAK меньше, чем 3.1.

Функция № 3: MassageAudio

Функция преобразует звуковые данные в выходной формат аппаратного обеспечения. Данные будут преобразованы в необходимый аппаратно-зависимый формат в соответствии с заголовком структуры. Например, некоторые устройства не позволяют воспроизводить звук

непосредственно в беззнаковом 8-битном формате. После переформатирования содержимого памяти, вы можете без перезагрузки данных использовать функцию DigPlay2 столько раз, сколько это необходимо.

ВХОД: AX=68Ah

DS:SI *Реальный режим* Указатель на звуковую структуру, описывающую звуковой эффект.

ESI *Прямая адресация* В случае использования 32-разрядной адресации, ESI указывает на 32-разрядный адрес звуковой структуры в первом мегабайте адресного пространства. Будет преобразован драйвером DIGPAK в допустимый адрес СЕГМЕНТ:СМЕЩЕНИЕ.

Функция № 4: DigPlay2

Функция исполняет звуковые данные, преобразованные в необходимый аппаратно-зависимый формат. В сочетании с MassageAudio она позволяет воспроизводить звуковые данные, не изменяя содержимого звукового буфера.

ВХОД: AX-68ВП

DS:SI * Реальный режим* Указатель на звуковую структуру, описывающую звуковой эффект.

ESI *Прямая адресация* В случае использования 32-разрядной адресации, ESI указывает на 32-разрядный адрес звуковой структуры в первом мегабайте адресного пространства. Будет преобразован драйвером DIGPAK в допустимый адрес СЕГМЕНТ:СМЕЩЕНИЕ.

ВЫХОД: Ничего

Функция № 5: AudioCapabilities

Данная функция сообщает о возможностях аппаратного обеспечения. Она возвращает информацию от резидентного драйвера DIGPAK об ограничениях и специфических особенностях платы в виде слова состояния, которое вы можете легко проанализировать.

ВХОД: AX=68Ch ВЫХОД: AX=Бит 0 -> 1 - поддерживается фоновое воспроизведение

0 - драйвер воспроизводит звук только как основной процесс.

Бит 1-> 1 - исходные данные должны быть преобразованы для выходного устройства;

0 - устройство способно воспроизводить неформатированные 8-битовые беззнаковые звуковые данные

Бит 2 -> 1 - устройство может воспроизводить звук только с фиксированной частотой, но звуковой драйвер способен понизить частоту входных данных

0 - устройство способно воспроизводить звук с указанной пользователем частотой.

(Замечание: Вы в любом случае можете воспроизводить звуковой фрагмент с нужной частотой, драйвер способен понизить частоту входных данных. Однако повышение частоты пока не поддерживается.)

Бит 3 -> 1 — устройство использует вектор прерывания таймера во время воспроизведения звука.

Бит 4 -> 1 — драйвер поддерживает разделение таймера (не доступно в защищенном режиме).

Бит 5-> 1 — поддерживается режим ожидания.

Бит б-> 1 — поддерживается режим стереопанорамы.

Бит 7-> 1 - поддерживается воспроизведение 8-битного PCM-стереозвука.

Бит 8-> 1 — поддерживается запись звука.

Бит 9-> 1 - поддерживается режим обратной записи DMA.

DX = Если драйвер может воспроизводить данные только с определенной частотой, в регистре DX содержится значение этой частоты.

Функция № 8: StopSound

Функция немедленно останавливает воспроизведение музыкального фрагмента.

ВХОД: AX=68Fh

ВЫХОД: Ничего Вызывает немедленное прекращение воспроизведения текущего музыкального фрагмента.

Функция № 12: SetTimerDivisorRate

Эта функция не работает в защищенном режиме. Она позволяет прикладной программе информировать драйвер DIGPAK, использующий таймер, о перепрограммировании частоты прерываний таймера 8253. Данная функция используется только в реальном режиме, так как ни один из базирующихся на прерывании таймера драйверов DIGPAK не работает в защищенном режиме. Если ваша компьютерная игра перепрограммировала таймер 8253 на новую частоту, вы должны сообщить об этом драйверу DIGPAK. Для того чтобы обеспечить воспроизведение

музыкального фрагмента, драйвер DIGPAK, использующий таймер, должен перепрограммировать 8253 на очень высокую частоту- По умолчанию DIGPAK считает исходную частоту прерываний таймера равной стандартной (18.2 раза в секунду) и устанавливает это значение после завершения музыкального эффекта. Однако данная функция позволяет вам поставить драйвер DIGPAK в известность об изменении базовой частоты. В результате этого драйвер будет обслуживать программу прерываниями таймера с указанной частотой, а после воспроизведения звукового эффекта драйвер установит таймер на эту частоту.

ВХОД: AX=693h

DX= Делитель частоты таймера. Драйверы, базирующиеся на прерываниях от таймера, способны обслуживать прикладную программу прерываниями от таймера с заданной делителем частотой независимо от частоты прерываний, необходимой для воспроизведения звукового эффекта. Реально частота прерываний будет стремиться к заданной величине. Для восстановления стандартной частоты прерываний (18.2 раза в секунду) в регистре DX следует задать 0.

ВНИМАНИЕ!!! Если вы перепрограммируете таймер в своей программе, при выходе из нее обязательно восстановите нормальное значение частоты прерываний — 18.2 раза в секунду.

ВЫХОД: Ничего

Функция № 14: PostAudioPending

Данная функция позволяет вам поместить в очередь второй музыкальный фрагмент. Начиная с версии DIGPAK 3.0, приложение может задать второй музыкальный фрагмент, который должен быть воспроизведен сразу по окончании воспроизведения текущего фрагмента. Этот метод, называемый двойной буферизацией, позволяет вашей прикладной программе осуществлять очень сложное манипулирование звуком. Используя двойную буферизацию, вы можете производить программное микшироваиие и (или) подкачивать длинное музыкальное произведение с диска, или из расширенной памяти EMS. В демонстрационной программе PEND.C показано, как можно исполнить по частям большой звуковой клип. Модифицируя эту демонстрационную программу, вы можете воспроизводить звуковые эффекты с диска, или выполнять программное микширование, добавляя набор звуковых эффектов в основной музыкальный поток.

ВХОД: AX=695h

DS:SI *Реальный режим*. Указатель на звуковую структуру, описывающую звуковой эффект.

ESI *Прямая адресация* В случае использования 32-разрядной адресации, ESI указывает на 32-разрядный адрес звуковой структуры в первом мегабайте адресного пространства. Будет преобразован драйвером DIGPAK в допустимый адрес СЕГМЕНТ:СМЕЩЕНИЕ.

ВЫХОД: AX=0 Началось воспроизведение фрагмента.

AX=1 Фрагмент поставлен в очередь на воспроизведение.

AX=2 В очереди уже имеется фрагмент, поэтому данный фрагмент в очередь не поставлен.

Функция № 15: AudioPendingStatus

Функция позволяет получить текущий статус состояния двойного буфера функции PostAud io Pending. Система с двойной буферизацией может находиться в одном из трех состояний. В состоянии 0 никакой звуковой фрагмент не воспроизводится. В состоянии 1 музыкальный фрагмент воспроизводится, но в очереди фрагментов больше нет (это означает, что пришло время поместить- в очередь следующий звуковой фрагмент). И, наконец, последний вариант — первый фрагмент воспроизводится, а второй уже находится в очереди, в этом случае никаких действий от прикладной программы не требуется,

ВХОД: AX=696Ь ВЫХОД: AX=0 Никакой фрагмент не воспроизводится.

AX=1 Один фрагмент воспроизводится, другой фрагмент ожидает своей очереди.

AX=2 Звуковой фрагмент воспроизводится, но очередь пуста.

Функция № 16: SetStereoPan

В настоящее время эта функция поддерживается только несколькими устройствами, например, Sound Blaster Pro. Вы можете вызывать AudioCapabilities для проверки возможности использования текущим драйвером режима стереопереходов. На некоторых стереозвуковых платах, таких как Sound Blaster Pro, вы можете управлять раскладкой монозвукового эффекта по стереоканалам. С помощью вызова данной функции вы указываете звуковой плате процентное •соотношение громкости вашего фрагмента для правого и левого динамиков. Вы можете вызывать эту функцию когда угодно, даже во время воспроизведения, чтобы перемещать звук от динамика к динамику. Это позволяет вам разнообразить вашу игру захватывающими и потрясающими эффектами.

ВХОД: AX=697h

DX= Процентное соотношение громкости по каналам:

0 — 100% на правый динамик.

64 — 50% по обоим динамикам.

127 — 100% на левый динамик.

ВЫХОД: AX=0 Команда проигнорирована, драйвер не поддерживает стереопереходов.

AX=1 Команда выполнена.

Функция № 17: SetPlayMode

функция позволяет установить режим записи/воспроизведения для резидентного драйвера DIGPAK. Из-за того, что, появившись в 1987 году, DIGPAK обеспечивал всего лишь одну функцию (воспроизведение звукового эффекта), независимая функция была наиболее простым способом расширения возможностей драйвера с одновременным сохранением совместимости сверху вниз. С помощью данной функции вы можете задать разрешение, при котором будет выполняться воспроизведение или запись звукового фрагмента.

ВХОД: AX=698h

DX== Режим воспроизведения. DX =0 -> 8-разрядный PCM.

=1 -> 8-разрядный стерео-PCM (правый/левый).

=2 -> 16-разрядный PCM.

=3 -> 16-разрядный стерео-PCM.

После изменения режима воспроизведения все функции DIGPAK работают точно так же, как и раньше, но ожидают данные в установленном формате. 8-разрядный стерео-PCM состоит из пар беззнаковых данных для обоих каналов. Поддерживается платами Stereo FX и Sound Blaster Pro. Все 16-разрядные данные являются знаковыми и в случае стереозвука их знак определяет правый или левый канал записи/воспроизведения.

ВЫХОД: AX = 1 Режим установлен.

AX = 0 Режим не поддерживается данным драйвером.

Функция № 18: Адрес флага ожидания и адрес «семафора» DIGPAK

Эта функция сообщает прикладной программе адрес в памяти, по которому расположен внутренний флаг ожидания драйвера DIGPAK. Этот метод называется «семафором и позволяет программе постоянно отслеживать состояние флага, не выполняя вызова функции. По изменению состояния этого флага, ваша программа может принимать решение о помещении следующего звукового фрагмента в очередь ожидания.

```
int far *ReportPendingAddress(void);
```

Сообщает дальний адрес флага ожидания. Если ячейка памяти по этому адресу содержит значение 1, это означает, что следующий фрагмент все еще ожидает своей очереди. Когда значение ячейки становится равным 0, ваша программа может начать загрузку следующего фрагмента в двойной буфер. Использование «семафора» для определения момента загрузки следующего фрагмента предпочтительнее, чем вызов функции AudioPendingStatus.

```
int far *ReportSemaphoreAddress(void) ;
```

Сообщает адрес «семафора» DIGPAK. Возвращаемый указатель является дальним указателем на ячейку памяти внутри драйвера DIGPAK. Если значение не нулевое, то DIGPAK находится в активном состоянии и вы не должны в это время вызывать какие бы то ни было функции DIGPAK по аппаратному прерыванию, так как это, возможно, прервет исполнение кода самого драйвера.

ВХОД: AX=699h

ВЫХОД: AX:DX Дальний адрес флага ожидания.

BX:DX Дальний адрес «семафора» DigPak. (При использовании DIGPAK.ASM в 32-разрядном режиме адресации, возвращаемый адрес будет преобразован в допустимый 32-разрядный адрес.)

Функция № 19: Установка режима записи

Функция позволяет вам установить DIGPAK в режим записи или воспроизведения звука. Вызов функции DigPlay после установки режима записи, обеспечивает запись звука в указанный буфер памяти. Функция поддерживается только для карт Sound Blaster.

ВХОД: AX-69AI1

DX=0 Включить режим записи.

DX=1 Выключить режим записи.

ВЫХОД: AX=0 Драйвер не поддерживает запись звука.

AX=1 Режим записи звука установлен.

Функция № 21: Установка режима обратной записи DMA

Функция позволяет включить режим обратной записи DMA. В идеале было бы достаточно использовать функцию PostAudioPending. Однако большинство моделей звуковых карт не позволяют гладко проиграть две звуковые последовательности одну за другой. И хотя фирма Creative Labs уже продала свыше двух миллионов карт Sound Blaster без этого недостатка, другие звуковые платы по-прежнему не способны обеспечить ровную стыковку двух звуковых фрагментов. Единственный способ обойти данную проблему состоит в использовании метода обратной записи DMA. В идеале функция PostAudio Pending должна была бы автоматически использовать этот метод, но это заняло бы слишком много времени для изменения всех имеющихся драйверов. Метод, с помощью которогб DIGPAK поддерживает режим обратной записи, достаточно прямолинеен, но пока я придумываю что-нибудь получше, вы вполне можете им воспользоваться.

Посмотрев на демонстрационную программу PEND.C, вы увидите пример того, как вместо PostAudioPending используется метод обратной записи DMA в том случае, если это поддерживается звуковой картой. Вы можете свободно использовать этот подход к реализации двойной буферизации в своих программах.

Обычно во время воспроизведения музыкального фрагмента звуковая карта осуществляет прямой доступ в память, который прерывается программой по окончании воспроизведения звука. Такой метод используется для исполнения одного фрагмента за один раз и это является стандартным режимом для вызовов функций DIGPAK. При автоматической инициализации прямого доступа в память, по достижении конца звукового фрагмента контроллер прямого доступа в память немедленно возвращается в начало буфера; он никогда самостоятельно не заканчивает воспроизведение. Следовательно, если ваш фрагмент — это слово «Привет!» и вы инициировали прямой доступ в память, то будете слышать «Привет!» непрерывно до тех пор, пока не прервете воспроизведение.

Как же использовать подобное поведение контроллера прямого доступа в память для двойной буферизации? Вначале вы должны выбрать фиксированный размер буфера передачи, например, 2 килобайта. Затем, если ваш буфер размером 2K, вы должны зарезервировать двойной буфер, то есть 4K. Теперь, начав воспроизведение этого четырехкилобайтного буфера, вы должны следить за тем, какую половину буфера в данный момент проигрывает DIGPAK. Каждый раз, когда DMA пересекает границу 2K, вам следует записать следующие 2K позади текущей позиции воспроизведения! Вот почему этот метод получил название обратной записи. Фактически, вы записываете данные позади текущей передачи данных с использованием прямого доступа в память. Чтобы жизнь не показалась вам такой уж простой, помните, что вы не можете использовать любые 4K для подобного буфера. Он не может пересекать границ страницы. Границы страниц располагаются через каждые 64K в первом мегабайте адресного Пространства компьютера. Поэтому DIGPAK содержит функцию, которая позволяет выяснить, не пересекает ли выбранный вами буфер одну из таких границ. если да, то просто выберите следующие 4K для вашего буфера, они уже точно пересекут границу.

Все это звучит достаточно запутано... но так оно и есть. Именно поэтому я и предоставил в ваше распоряжение простую программу PEND.C. Вы можете использовать ее как прообраз для своих функций воспроизведения звука. Используя прямой доступ в память и двойную буферизацию, вы сможете подкачивать звуковые данные с диска, осуществлять программное микширование в реальном времени и создавать звуковое сопровождение для фильмов и FLIC-файлов.

ВХОД: AX = 69СЬ

DX=0 Выключить режим обратной записи.

DX=1 Включить режим обратной записи.

ВЫХОД: AX=0 Режим обратной записи не поддерживается драйвером.

AX=1 Режим обратной записи установлен.

Функция № 22: Значение счетчика прямого доступа в память

Функция возвращает текущее значение счетчика при прямом доступе в память. Другими словами, если вы начали передачу из буфера в 4K, в процессе воспроизведения счетчик будет изменяться от 4097 до 0. Отслеживая значения счетчика, вы можете определить момент, когда следует заполнять следующую половину буфера. В программе PEND.C показано, как это сделать на практике.

ВХОД: AX=69Dh

ВЫХОД: AX= Текущее значения счетчика (значение убывает, а не возрастает).

Функция № 23: Проверка буфера прямого доступа в память

Функция позволяет проконтролировать, не пересекает ли выбранный вами буфер прямого доступа в память грашщ страницы. Пример использования этой функции приведен в программе PEND.C.

ВХОД: AX=69Eh

ES:BX *Реальный режим* Указатель на звуковую структуру, описывающую звуковой эффект.

EBX *Прямая адресация* В случае использования 32-разрядной адресации, EBX указывает на 32-разрядный адрес звуковой структуры в первом мегабайте адресного пространства. Будет преобразован драйвером DIGPAK в допустимый адрес СЕГМЕНТ: СМЕЩЕНИЕ.

CX = Размер буфера. ВЫХОД: AX=1 Блок не пересекает границу страницы. AX=0 Блок пересекает границу страницы.

Функция № 24: Установка громкости при импульсно-кодовой модуляции

Функция, реализована только для нескольких звуковых карт. Она позволяет установить для карты общую относительную громкость звука.

ВХОД: AX=69Fh

BX = Громкость левого канала (или обоих для моно) 0-100.

CX = Громкость правого канала (или обоих для моно) 0-100.

ВЫХОД: AX=1 Громкость установлена.

AX=0 Устройство не поддерживает установку громкости.

Функция № 25: SetDPMIMode

Функция информирует драйвер DIGPAK о 32-разрядной адресации. Поскольку оригинальные драйверы DIGPAK предполагалось использовать только в реальном режиме, я не задумывался о возможности другой адресации. (Первые драйверы DIGPAK разрабатывались в 1987 году, поэтому в этом нет ничего удивительного.) Из-за того, что многие функции DIGPAK используют комбинацию регистров сегмента и смещения для формирования адреса, это порождает определенные проблемы в защищенном режиме. В защищенном режиме сегментные регистры являются селекторами и сформировать из них адрес реального режима трудно. После вызова этой функции, DIGPAK будет знать, что он активизирован прикладной программой для защищенного режима и будет реагировать на полный 32-разрядный адрес в ESI, а не на комбинацию DS:SI. Процесс трансляции адреса автоматически обеспечивается интерфейсом нижнего Уровня DIGPLAY.ASM, находящимся в подкаталоге программ для защищенного режима.

ВХОД: AX=6A0h

DX= Режим включен/выключен (1/0). ВЫХОД: Ничего

```
#*****************************************************************
#***** DigPlay, компонуемый интерфейс Digitized Sound Package ****
#*****************************************************************
DIGPLAY.H Компонуемый интерфейс реального режима. Все процедуры имеют прототипы и
используют имена сегментов, пригодные для любой модели памяти.

#ifndef LOADABLE_DRIVERS
ftdefine LOADABLE_DRIVERS 1// условная компиляция
#endif
/* битовые флаги возможностей драйвера     */
/* возвращаются функцией AudioCapabilities */
#define PLAYBACK       1   // бит 0, возможно воспроизведение звука
//  в фоновом режиме
#define MASSAGE        2
// бит 1, данные преобразованы
//в аппартно-эависимый формат
#define FIXEDFREQ      4   // бит 2, драйвер воспроизводит звук
//  только с фиксированной частотой
#define USESTIMER      8   // бит 3, драйвер использует таймер
#define SHARESTIMER   16   // бит 4, таймер может быть использован
//  совместно (бета-версия!!!)
#define LOOPEND       32   // бит 5, поддерживается цикличность и
//  очередь ожидания (бета-версия!!!)
#define STEREOPAN     64   // бит 6, поддерживаются стереопереходы
#define STEREOPLAY   128   // бит 7, поддерживается 8-разрядный
                        //  PCM-звук
#define AUDIORECORD  256   // бит 8, поддерживается запись звука
#define DMABACKFILL  512   // бит 9, поддерживается режим обратной
// записи DMA
#define PCM16      1024   // бит 10, поддерживается 16-разрядный
//  звук
#define PCM16STEREO 2048   // бит 11, поддерживается 16-разрядный
//  стереозвук
typedef struct
{ char far *sound;            // адрес звуковых данных
unsigned short sndlen;     // длина звуковых данных
short far *IsPlaying;       // адрес флага состояния
short frequency;            // частота воспроизведения }SNDSTRUC;
extern short far cdecl DigPlay (SNDSTRUC far *sndplay) ;
// 688h -> воспроизведение 8-разрядного оцифрованного звука
extern short far cdecl SoundStatus (void);
// 689h -> сообщает состояние звукового драйвера
extern void far cdecl MassageAudio (SNDSTRUC far *sndplay);
// 68Ah -> преобразует цифровой звук в аппаратнозависимый формат
extern void far cdecl DigPlay2 (SNDSTRUC far *sndplay);
// 6SBh -> воспроизведение звука в аппаратноэависимом формате
extern short far cdecl AudioCapabilities (void);
// 68Ch -> сообщает информацию о возможностях драйвера
extern short far cdecl DigPakIdentityString (char far *str) ;
// 68Ch -> копирует в заданный буфер идентификатор драйвера
// звука и возвращает длину скопированной строки
```

```
extern void far cdecl StopSound (void) ;
// 68Fh -> останавливает воспроизведение звука
extern short far cdecl PostAudioPending (SNDSTRUC far *sndplay);
#define NOTPLAYING 0 // звук не воспроизводится
#define PLAYINGNOTPENDING  1 // звук воспроизводится, очередь пуста
#define PENDINGSOUND       2 // звук воспроизводится, следующий
//  фрагмент ожидает своей очереди
extern short far cdecl AudioPendingStatus (void);
#define FULLRIGHT           0
#define FULLLEFT            127
#define FRONTANDCENTER      64
extern short far cdecl SetStereoPan(short panvalue);
// 0 - 127, 0 - 100% правый канал
#define PCM_8_MONO      0
#define PCM_8_STEREO   1
#define PCM_16_MONO 2 #define PCM_16__STEREO        3
extern short far cdecl SetPlayMode (short playmode);
// 0 - режим не поддерживается
// 1– режим поддерживается
extern short far cdecl SetRecordMode(short mode);
// устанавливает режим записи звука
extern short far * far cdecl PendingAddress (void);
// сообщает дальний адрес флага ожидания. Если ячейка памяти по
// этому адресу содержит значение 1, это означает, что следующий
// фрагмент все еще ожидает своей очереди. Когда значение ячейки
// становится равным 0, ваша программа может начать загрузку
// следующего фрагмента в двойной буфер. Использование "семафора"
// для определения момента для загрузки следующего фрагмента
// предпочтительнее, чем вызов функции AudioPendingStatus.
extern short far * cdecl ReportSemaphoreAddress(void);
// сообщает адрес семафора DIGPAK. Возвращаемый указатель является
// дальним указателем на ячейку памяти внутри драйвера DIGPAK. Если
// значение не нулевое, то DIGPAK находится в активном состоянии и
// вы не должны в это время вызывать какие бы то ни было функции
// DIGPAK по аппаратному прерыванию, так как это, возможно, прервет
// исполнение кода самого драйвера.
extern void far cdecl SetTimerDivisorRate (short rate) ;
// устанавливает делитель частоты для таймера
// Если ваша программа перепрограммировала таймер 8253 на новую
// частоту, вы должны сообщить об этом драйверу DIGPAK.
// Не забудьте восстановить стандартную частоту после завершения
// работы программы.
extern short far cdecl ReportVersionNumber (void);
// сообщает версию драйвера DIGPAK
// Номер версии возвращается умноженным на сто, то есть значение
// 310 соответствует версии 3.1. Драйверы версий младше 3.1 не
// поддерживают данную функцию, поэтому нулевое значение означает,
// что версия загруженного драйвера DIGPAK меньше, чем 3.1.
extern short far cdecl SetBackFillMode (short mode);
// Устанавливает режим обратной записи DMA. Возвращает 1, если
// режим.установлен, и 0, если драйвер не поддерживает этот режим.
extern unsigned short far cdecl ReportDMAC (void);
// сообщает текущее значение счётчика DMA
extern short far cdecl VerifyDMA (chac far *data, short length);
// Проверяет, не пересекает ли буфер границ страницы. Возвращает 1,
// если граница не пересекается, и 0, если буфер пересек границу.
extern void far cdecl NullSound (char far *sound, short sndlen, short null) ;
/* Дополнительные функции */
extern void far cdecl WaitSound (void);
// ожидание окончания воспроизведения звука
extern short far cdecl Checkin (void);
// Загружён ли драйвер. 1 - да; 0 - нет.
/**** Внимание!!! Если вы используете LOADABLE_DRIVERS, то должны
обеспечить доступ к функциям распределения памяти и к DOSCALLS.OBJ.
extern short far cdecl InitDP (short segment);
 // инициализировать драйвер DIGPAK
extern void far cdecl DeInitDP (short segment);
```

```
// выгрузить драйвер DIGPAK
```

API пакета программ MIDPAK

Драйвер MIDI, MIDPAK использует вектор прерывания 66h, что обеспечивает прозрачный программный интерфейс. Ниже.описываются два способа воспроизведения MIDI-музыки. Первый способ основан на прерываниях. Второй — на использовании библиотеки функций на Си или ассемблере, которые позволяют не только осуществлять доступ к драйверам звука, но и предоставляют другие полезные функции. Эти функции находятся в исходном файле MIDPACK.ASM.

MIDPAK использует тот же вектор прерывания, что и DIGPAK. Пакет DIGPAK описывает полный набор драйверов оцифрованного звука, поставляемый фирмой Audio Solution. MIDPAK полностью совместим с DIGPAK. Если ваша программа должна воспроизводить как MIDI-музыку, так и оцифрованный звук, надо вначале загрузить требуемый драйвер оцифрованного звука, и затем поверх него загрузить MIDI-драйвер MIDPAK. Драйвер MIDPAK обнаруживает присутствие драйвера DIGPAK и перенаправляет через него все вызовы. Если аппаратное обеспечение не в состоянии независимо воспроизводить оцифрованный звук (поддержка прямого доступа в память: Sound Blaster и ProAudio Spectrum), тогда во время воспроизведения оцифрованного звука исполнение MIDI-музыки будет выключено. Воспроизведение MIDI-музыки возобновится сразу же после окончания проигрывания оцифрованного звукового фрагмента. Для прикладной программы этот процесс полностью прозрачен.

Пакет программ MIDPAK использует набор звуковых драйверов MIDI, разработанных Miles Design Inc. Эти драйверы различаются по своему размеру и имеют расширение .ADV. При старте MIDPAK всегда загружает драйвер MUSIC.ADV. Поэтому прикладная программа перед загрузкой MIDPAK должна переименовать необходимый драйвер в MUSIC.ADV.

MIDPAK не исполняет непосредственно файлы MIDI. Вы должны конвертировать файлы MIDI (с расширением .MID) в файлы обобщенного MIDI (-XMI), используя программу MIDIFORM или утилиту MENU. Драйверы Расширенного MIDI фирмы Miles Design Incorporated поддерживают каналы 2-9 для мелодических инструментов и канал 10 для ударных.

Замечание

Любой не указанный номер функции является устаревшим или не используется.

Функция № 1: UnloadMidPak

Эта функция освобождает память, занятую резидентной частью MIDPAK, и на должна использоваться прикладной программой! Используется MIDPAK для внутренних целей и приведена здесь только для полноты картины.

ВХОД: AX=700h Номер команды.
ВЫХОД: Ничего

Функция № 2: DigPakAvailable

Функция определяет доступность драйвера DIGPAK под драйвером MIDPAK.
ВХОД: AX=701h Номер команды.
ВЫХОД: AX=0 DIGPAK не доступен.
AX=1 DIGPAK доступен.

Функция № 3 PlaySequence

Функция исполняет последовательность из текущего зарегистрированного XMIDI-файла.
ВХОД: AX=702h Номер команды,
BX=SEQ Номер последовательности, начиная с нуля,
ВЫХОД: AX=1 Последовательность проигрывается.
AX=0 Последовательность не доступна.

Функция № 4: SegueSequence

Функция регистрирует с указанием кода активации новую последовательность Для исполнения по триггерному событию. Если значение кода активации -1, то переход к данной последовательности будет осуществлен по ближайшему триггеру. Триггер с указанием кода события помещается в поток данных MIDI с помощью Контроллера 119. Контроллеры 119 могут быть помещены в любое место потока данных MIDI для передачи программе информации о текуацей позиций в MIDI-последовательности.

ВХОД: AX=703h Номер команды.
BX = SEQ Номер регистрируемой последовательности.
CX= ACT Код активации события, -1 означает следующий триггер.

Функция № 5: RegisterXmidi

Функция регистрирует адрес файла XMIDI для исполнения.
ВХОД: AX=704h Номер команды,
BX=Offset Смещение в дальнем адресе данных XMIDI.
CX=Segment Сегмент в дальнем адресе данных XMIDI.
SI=Low len Младшее слово значения длины данных XMIDI.
DI=High len Старшее слово значения длины данных XMIDI.
ВЫХОД: AX=0 Ошибка регистрации данных XMIDI.
AX=1 Файл XMIDI зарегистрирован резидентно. Это означает, что файл полностью поместился во внутренний буфер MIDPAK. Ваша программа может освободить память, связанную с файлом XMIDI, так как MIDPAK создал для себя его, копию. Это очень полезно в средах с виртуальной памятью, где прикладная программа не всегда имеет фиксированный адрес в памяти. Это также позволяет MIDPAK исполнять музыку в фоновом режиме под DOS.
AX=2 Файл XMIDI зарегистрирован. Прикладная программа ответственна за то, чтобы указанный фиксированный адрес в памяти всегда содержал соответствующие данные XMIDI.

Функция № 6: MidiStop

Функция, останавливает воспроизведение текущей последовательности MIDI.
ВХОД: AX=705h Номер команды.
ВЫХОД: Ничего

Функция № 8: ReportTriggerCount

Функция возвращает счетчик триггерных событий и код последнего события.
ВХОД: AX=707h Номер команды.
ВЫХОД: AX=COUNT Количество событий со времени последнего сброса счетчика.
DX=ID Код последнего события. Коды событий вы можете найти в спецификации XMIDI..

Функция № 9: ResetTriggerCount

Функция сбрасывает счетчик событий в ноль.
ВХОД: AX=708h Номер команды.
ВЫХОД: Ничего

Функция № 12: ResumePlaying

Функция продолжает исполнение остановленной последовательности.
ВХОД: AX-70Bh Номер команды.

Функция № 13: SequenceStatus

Функция возвращает состояние последовательности.
ВХОД: AX=70Ch Номер команды.
ВЫХОД: AX=Статус
SEQ_STOPPED 0 Воспроизведение последовательности остановлено.
SEQ_PLAYING 1 Последовательность исполняется в настоящий момент.
SEQ_DONE 2 Исполнение последовательности завершено.

Функция № 14: RegisterXmidiFile

Функция регистрирует файл по его имени.
ВХОД: AX=70Dh Номер команды.
BX=Offset Смещение адреса имени файла.
CX=Segment Сегмент адреса имени файла.

Функция № 15: RelativeVolume

Функция возвращает относительную громкость музыки в процентах.
ВХОД: AX=70Eh Номер команды.
ВЫХОД: AX= VOL Текущая относительная громкость в процентах 0-100.

Функция № 16: SetRelativeVolume

Функция устанавливает относительную громкость в процентах.

ВХОД: AX=70Fh Номер команды.

Функция № 17: BootstrapMidPak

Функция позволяет приложению устанавливать драйвер MIDPAK.

ВХОД: AX-710h Номер команды.

BX:CX СЕГМЕНТ:СМЕЩЕНИЕ драйвера ADV.

DX:SI СЕГМЕНТ:СМЕЩЕНИЕ файла AD.

Функция № 18: PollMidPak

ВХОД; AX=710h

Функция используется в сочетании с PMIDPAK.COM. Это версия MIDPAK, работающая по опросу. Обычный MIDPAK перехватывает прерывание либо от таймера, либо от часов реального времени и обслуживает его с частотой 120 раз в минуту. Однако некоторые сложные прикладные программы сами обслуживают аппаратные прерывания и требуют исполнения фоновой музыки с иной частотой дискретизации, или синхронизируют по таймеру графические функции, чтобы избежать возможного прерывания музыкой графических процедур.

После того как MIDPAK установлен, он восстанавливает вектор прерывания таймера и не проявляет себя до- тех пор, пока программа, не выполнит прерывание 66h с командой 0711h. Вы должны вызывать это прерывание с частотой 120 раз в минуту или с частотой дискретизации, указанной при запуске MIDIFORM. При запуске MIDIFORM вы можете указать частоту дискретизации для вашей музыки, отличную от принятого по умолчанию значения 120. При снижении частоты дискретизации вы услышите ухудшение качества музыки, так как ASDR не будет реагировать достаточно быстро. Снижение ее, например, до 60 раз в минуту не будет сильно заметным, однако уменьшение частоты до 30 или 15 вызовет значительное ухудшение качества звучания. Очевидно, что исполнение многоканальной MIDI-музыки на таком частотном синтезаторе, как Adiib, потребует определенных ресурсов процессора. Переквантовывая свою музыку и задавая MIDPAK удобные для вас частоты, вы можете добиться хорошего баланса использования ресурсов компьютера.

Функция № 19: MidpakClock

Функция возвращает текущее значение счетчика MIDPAK.

ВХОД: AX=7l2h

ВЫХОД: AX:DX Текущее значение счетчика с момента старта MIDPAK. Счетчик обновляется 120 раз в минуту, и ваше приложение может использовать его как таймер.

BX:CX Образует дальний указатель на счетчик.

BX Смещение, CX Сегмент.

Функция возвращает значение внутреннего счетчика MIDPAK. При старте счетчик, представляющий собой двойное слово, имеет нулевое значение, которое увеличивается каждую минуту на 120, и ваше приложение может использовать его как таймер, просто опрашивая эту функцию.

Функция № 20: TriggerCountAddress

Функция возвращает адрес размещения счетчика триггеров.

ВХОД: AX=713h

ВЫХОД: AX:DX Образует адрес целого значения счетчика триггеров. Значение этого счетчика увеличивается на единицу при каждом появлении Контроллера 119 в файле MIDI.

Функция № 21: EventIDAddress

Функция возвращает адрес размещения кода события.

ВХОД: AX=714h

ВЫХОД: AX:DX Образует адрес целого значения кода события. В этой ячейке памяти хранится код последнего события вызванного Контроллером 119.

Функция № 23: ReportSequenceNumber

Функция возвращает номер исполняемой в данный момент последовательности.

ВХОД: AX=716h

ВЫХОД: Возвращает номер исполняемой в данный момент последовательности.

```
extern short cdecl CheckMidiIn (void);
// Возвращает 1, если MIDPAK установлен, 0 - если нет.
extern short cdecl DigpakAvailable (void) ;
// Возвращает 1, если DIGPAK установлен, 0 - если нет.
/***************************•**********************************
/**    Эти флаги возвращает функция регистрации данных XMIDI   ***
/*************************************************************/
```

```
#define FAILURE_TO_REGISTER   0 // Ошибка регистрации XMIDI файла.
#define REGISTERED RESIDENT   1 // Резидентный драйвер полностью
// содержит данные XMIDI. Приложение
// может освободить память, которую
// они занимали.
#define REGISTERED_APPLICATION 1 //Драйвер не имеет настолько
// большого буфера, чтобы полностью
// загрузить в него данные XMIDI.
// Приложение обязано обеспечить
// сохранение в памяти по указанному
// адресу данные XMIDI.
extern short cdecl PlaySequence (short seqnum);
// исполняет последовательность с данным номером
// из зарегистрированного файла XMIDI
#define NEXT_CALLBACK - 1 // активизация по ближайшему событию
extern short cdecl SegueSequence (short seqnum, short activate) ;
// Переключает исполнение последовательности на
// указанную последовательность по наступлению
// события с кодом, равным указанному коду активации.
// Если код активации равен -1, переключение
// произойдет по ближайшему событию.
extern short cdecl RegisterXmidi (char *xmidi, long int size) ;
// Регистрирует XMIDI файл для воспроизведения.
// Этот вызов зарегистрирует все последовательности.
extern short cdecl MidiStop (void);
// остановка воспроизведения текущей последовательности
extern long int cdecl ReportCallbackTrigger (void);
// младшее слово - счетчик триггеров
// старшее слово - идентификатор последнего события
extern void cdecl ResetCallbackCounter (void);
// сбрасывает счетчик триггеров в ноль
extern void cdecl ResumePlaying (void) ;
// продолжает воспроизведение прерванной последовательности
#define SEQ_STOPPED  0  // возвращаемые значения
#define SEQ_PLAYING  1  // функции SequenceStatus()
#define SEQ_DONE     2
extern short cdecl SequenceStatus (void) ;
// возвращает состояние текущей последовательности
extern short cdecl RelativeVolume (short vol);
// возвращает текущую громкость
extern void cdecl SetRelativeVolume (short vol, short time);
// устанавливает громкость на заданный период времени
#define NOBUFFER          1  // нет резидентного буфера
#define FILENOTFOUND      2  // файл не найден
#define FILETOBIG         3  // файл превышает размер
// зарезервированного буфера
#define REGISTRATIONERROR 4  // ошибка регистрации файла XMI
extern short cdecl RegisterXmidiFile (char *fname);
// регистрирует файл по имени
extern void cdecl PollMidPak (void);
           // запрос MIDPAK на исполнение музыки
extern long int cdecl MidPakClock (void);
// возвращает значение внутреннего счетчика MIDPAK
extern long int * cdecl MidPakClockAddress (void);
// возвращает адрес таймера MIDPAK
extern short * cdecl TriggerCountAddress (void) ;
// возвращает адрес счетчика триггеров
extern short * cdecl EventIDAddress (void);
// возвращает адрес идентификатора события
extern short cdecl ReportSequenceNumber (void) ;
extern short cdecl InitMP (char *midpak, char *adv, char *ad) ;
// инициализирует драйвер MIDPAK
extern void cdecl DeInitMP (char *midpak);
// выгружает драйвер MIDPAK
```

Завтрак с Толстяком - музыка в программах

Я попросил моего друга, Джорджа Алистара Зангера, известного под псевдонимом Толстяк,

порассуждать о том, как, когда и почему музыка должна использоваться в программах. Толстяк — популярный композитор, предоставляющий широкий диапазон услуг в индустрии мультимедиа.

Толстяк создает музыку в своем доме в Остине, штат Техас для клиентов от Лос-Анджелеса до Нью-Йорка и Гонконга. Среди его клиентов Warner/Elektra/Atlantic, Cannon Films и Southwest Airlines. Он участвовал в создании музыкального оформления для десятка программных продуктов, включая Wing Commander фирмы Origin Systems, The 7th Guest фирмы Virgm/Trilobyte и SSN-21 Seawolf фирмы Electronic Arts.

Почему?

Почему музыкальные и звуковые эффекты должны использоваться в компьютерных программах? По той же самой причине, по которой все окружающие пользуются пиктограммами, — потому что это следующий логический шаг в разработке компьютерных программ. Пиктограммы и графика избавляют пользователя от необходимости представлять себе, что же они должны увидеть, музыка же говорит им, что они должны почувствовать.

Я бы не хотел принижать интеллект среднего пользователя, но чтение требует определенной аналитической работы. Чтение медленный процесс, оно не соответствует нашему естественному восприятию мира, и пользователи не очень-то любят читать. Вот тут на помощь и приходят пиктограммы. Уже установлено, что они повышают производительность, предлагая нам образы из «реального мира» (что бы это ни значило) и позволяют нам связать свои представления, например, о книжном шкафе или волшебнике с тем, что мы видим на экране. Музыка и звук продвигают все это на следующую логическую ступень.

Графика у пользователя ассоциируется с чувствами, вызываемыми объектом реального мира. Музыка и звук действуют еще более непосредственно. Они вызывают ассоциативные чувства — пользователь теперь уже может представить себе злого волшебника или удобный шкаф. И эти ассоциации могут оказаться очень мощным инструментом. В PC Magazine, в обзорной статье о мультимедиа, ветеран интерактивного видео предположил, что эмоции, вызываемые информацией, могут являться основной причиной информационных перегрузок и таким образом оказаться даже более важными, чем сама информация.

Звук и музыка могут:

- Повысить информативность;
- Сделать общение с компьютером более приятным;
- Повысить развлекательную ценность программы.

Когда?

Музыка и звук должны использоваться не только в играх.

Несмотря на то, что компьютерные программы Достаточно сильно отличаются от фильмов, разработчикам полезно быть в курсе последних новинок кинематографа и рассматривать фильмы как модели некоторых аспектов будущих компьютерных программ. Причем примеры можно найти не только в художественных фильмах, но и в рекламных роликах, учебных программах, новостях — в общем, во всем, что снимается на кино- или видеопленку.

Основное правило состоит в том, что музыка и звук используются тогда и только тогда, когда необходимо подчеркнуть эмоции, испытываемые зрителем. Следовательно, в случае компьютерных программ имеет смысл использовать музыку и звук в следующих ситуациях:

- Для усиления уже существующих эмоций. Многим нравится веселая рожица или что-либо подобное при запуске компьютера — с помощью веселой мелодии можно достигнуть того же эффекта. Логотип фирмы может быть подчеркнут звуковым эффектом. Подвижные игры могут рассматриваться как приключенческие фильмы, ну а стратегическим играм больше подойдет музыкальное оформление мистических или документальных лент;
- Для создания новых эмоций. Если вам действительно необходимо привлечь внимание пользователя, представьте себе, насколько звук клаксона будет эффективнее слова «ВНИМАНИЕ» в диалоговом окне. Писк при нажатии клавиши тоже для многих может сделать жизнь комфортнее. Если интересной графикой вы можете развеять скуку и пробудить интерес, то музыка может многократно усилить эти эмоции;
- Для манипулирования уже существующими эмоциями. Программа по детской безопасности может показывать изображение симпатичного ребенка возле бассейна. Тревожная мелодия поможет воспрепятствовать концентра ции внимания пользователя только на ребенке. Простая незамысловатая мелодия поможет сделать восприятие программы баз данных не таким пугающе сложным. В случае диалогового окна с «плохими новостями», насколько приятнее услышать приятный звуковой эффект, чем звук взрыва бомбы. С помощью звука разработчик так же, как режиссер в кино, можетконтролировать уровень эмоций, испытываемый пользователем.

Кроме того, не стоит использовать музыку и звук, если:

- Они вызывают не соответствующий обстоятельствам эмоциональный отклик. Это происходит, когда музыка не несет никакой смысловой нагрузки, просто плохо написана, или не адекватна задачам программного продукта. Такая музыка вызывает у пользователя лишь досаду и снижает ценность программного продукта;
- Они являются просто фоном.

Все это приводит нас к следующему вопросу...

Как?

Как и графика, музыка и звук должны быть хорошо написаны. Нельзя создать стоящую программу, просто заполнив свободное место на экране халтурной графикой, сделанной «другом-художником» (у всех нас есть такие). Абсолютно то же самое можно сказать и о музыке. На самом деле выбор подходящей музыки для программы ничуть не легче, чем для фильма. Но это слишком непростой вопрос, поэтому мы не будем на нем здесь останавливаться подробно.

Приведу лишь простое правило, воспользовавшись которым вы сможете себя проверить. Чтобы оценить качество вашего звука, вне зависимости от вашей аппаратной платформы, ответьте на три вопроса:

1. Поддерживает ли каждая нота эмоциональную направленность программы?
2. Повышает ли музыка интерес к программе?
3. Можете ли вы под нее танцевать?

Ну что ж, неплохо посидели. Надо бы встречаться почаще.

Создание пользовательского интерфейса с системой распознавания голоса

Я бы хотел немного остановиться на новых формах пользовательского интерфейса, базирующихся на звуке. Сейчас уже очевидно, что мы в состоянии использовать системы распознавания голоса в разработке интерфейса программы. Я бы хотел дать несколько общих советов тем, кто собирается включить распознавание голоса в свои прикладные программы.

Управление голосом является наиболее естественным способом общения с компьютером. Почему же до сих пор это не очень получается? Большинство современных голосовых систем пассивны; они работают на заднем плане, пытаясь угадать, что же вы хотели им сказать. Проявившись же, они стирают файл или делают что-нибудь похожее по идиотизму.

Большинство скажет, что проблема в самих программах распознавания голоса. Но это не так. Проблема в дизайне пользовательского интерфейса.

Интерфейс с распознаванием голоса должен быть активным. Он должен переспрашивать пользователя при сомнениях и подтверждать получение команды. Он так же должен персонифицироваться через технику общения и используемые выражения.

Попробуйте пообщаться с приятелем, который никогда не отвечает - у вас появится ощущение, что вас игнорируют. Как минимум, вам нужно иногда услышать «Да-а» или «Ага», чтобы знать, что вас слушают. Или возьмите другой пример. Отец просит ребенка что-то сделать, а в ответ не слышит:

«Хорошо, папа». В таком случае папа обычно начинает злиться и кричать: «Эй, ты меня слышишь?!» (так, по крайней мере, происходит в моем доме.) При разговоре нам необходима ответная реакция, чтобы знать, что нас поняли. Интерфейс с распознаванием голоса должен использовать реплики типа: «Конечно», «Хорошо, шеф», «Простите?»

Несколько подобных фраз могут сделать интерфейс более естественным и позволят нам управлять сложными системами набором нескольких простых голосовых команд. Сегодня не существует реальной проблемы с системами распознавания голоса. Обычно они распознают отдельные слова достаточно аккуратно. Существующие коммерческие программы позволяют нам совершать очень сложные действия, используя несколько меню и пиктограмм. Хорошо разработанная система с голосовым меню может быть не менее мощной и быть гораздо привлекательнее в использований.

И, наконец, последнее замечание о голосовом взаимодействии с компыотером. Говорящие машины заслужили плохую репутацию в начале 80-х из-за японских говорящих автомобилей: «Дверь открыта! Дверь открыта!». Все их ненавидели. Люди гордились тем, что им удалось «сломать» свои новые машины и выключить надоедливый голос. Заметьте, что проблема состояла не в плохом произношении машины, а в плохом пользовательском интерфейсе. Многих из нас еще в детстве научили, что не следует заговаривать с кем-либо, пока к вам не обратились. То же относится и к говорящим машинам. Если машина надоедает вам, прерывает ход ваших мыслей и мешает тому, что вы сейчас делаете, это очень раздражает. Однако если вы можете о чем-либо спросить машину и получить от нее ответ, это очень классная машина. Если машина хочет сообщить вам что-то действительно важное, она не должна врываться без приглашения, прерывая ваши мысли или разговор. Нет, она должна вежливо сказать «Гхм» или «Простите» либо просто прочистить свое электронное горло. Когда вы найдете время ответить и скажете: «Да, Саймон», компьютер может выдать накопившуюся информацию. Если системы с распознаванием голоса не являются достаточно естественными, они практически бесполезны. Мы будем продолжать использовать клавиатуру и мышь там, где это необходимо, но системы с распознаванием голоса должны стать более общительными, интерактивными и, самое главное, естественными.

Для персонификации машины одинаково подходит и использование разговорного языка, и акцент, и юмор. Обычно юмора боятся, вдруг кто-то подумает, что программа несерье;й1ая. Но кому понравится в очередной раз слушать идеальный, безжизненный голос «голосовой почты». Вместо этого, ваш компьютер мог бы сказать; «ОК, босс, все что пожелаете!» или «Конечно, Пап, нет проблем!» или «Инициализирую последовательность сохранения файла, доволен?» Конечно, это только примеры, и их не надо понимать буквально, просто они показывают, что общение с компьютером тоже может быть веселым. А ведь сделать общение с компьютером более естественным, простым, менее пугающим — и есть основная цель хорошо продуманного интерфейса.

В заключение, я хочу сказать, что приведенные здесь примеры представлены только для подстегиваиия вашей изобретательности. При разработке голосового интерфейса, просто задайте себе один вопрос: «Как бы это выглядело, если б я разговаривал с живым человеком?» Затем,

учитывая технические ограничения, сделайте нечто максимально похожее на живую речь. Помните, что хотя машина может «слышать» лишь ограниченное число слов, «сказать» же она может все, что угодно. Составьте простой набор команд, и сделайте его минимально подверженным ошибкам. Старайтесь по возможности учитывать пожелания пользователя. Пусть ваши голосовые ответы варьируются и, самое главное, будут естественными.

Демонстрационная программа Test.c

Ниже приводится демонстрационная программа на языке Си, которая показывает, как загружать и работать с драйверами DIGPAK и MIDPAK.

```
/*                                                            */
/* TEST.C Демонстрационная программа работы с драйверами DIGPAK */
/* и MIDPAK. Динамически загружает драйверы DIGPAK и MIDPAK,    */
/* SOUNDRV.COM и MIDPAK.COM/MIDPAK.ADV/MIDPAK.AD. Затем         */
/* воспроизводит MIDI-файл TEST.XMI, позволяя вам исполнять     */
/* звуковые эффекты TEST1.SND и TEST2.SND                       */
/**************************************************************/
/*      Автор: John W. Ratcliff (c) 1994                      */
/*               CompuServe: 70253,3237                       */
/*               Genie: J.RATCLIFF3                           */
/*               BBS: 1-314-939-0200                          */
/*               Адрес:                                       */
/*                         747 Napa Lane                      */
/*                         St. Charles, MO 63304              */
/*                                                            */
/*                                                            */
/*********************************************'****************/
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "keys.h"         // Включает определения для
// клавиатурных команд
#include "support.h"      // Включает файл заголовка базовых
// функций поддержки
#include "loader.h"       // Включает файл заголовка динамического
        // загрузчика MIDPAK/DIGPAK
#include "midpak.h"       // Включает файл заголовка функций
// нижнего уровня MIDPAK
#include "digplay.h"      // Включает файл заголовка функций
// нижнего уровня DIGPAK
#include "doscalls.h"     // Включает файл заголовка
// DOS-функций поддержки
#define NOBJ 4            // число загружаемых звуковых эффектов
static char *Names [NOBJ]= // имена файлов звуковых эффектов
{
"TEST1.SND", "TEST2.SND", "PEND.3ND", "TEST.SND" };
static SNDSTRUC *snd;        // Структура звуковых данных DIGPAK
static char *soundbuffer=0;  // Адрес буфера звуковых данных
static long int ssize[NOBJ]; // Длина загружаемого звукового эффекта
static int NALLOC=0;
char *Sounds[NOBJ];          // адреса всех звуковых эффектов
void UnloadSounds(void);     // Выгрузка звуковых эффектов из памяти
int  LoadSounds(void);       // Загрузка звуковых эффектов в память
void PlaySound(int sound);   // Воспроизведение звукового эффекта
void TestDigPak(void);       // Тестирование функций
// воспроизведения DIGPAK
// Приложение должно обеспечить функции резервирования памяти. Они
// используются загрузчиком и DOS функциями поддержки. Вы можете
// заменить эти функции любой другой системой управления памятью.
unsigned char far * far memalloc(long int siz) {
unsigned char far *mem;
mem = farmalloc(siz) ; // функция Си для резервирования
                  // памяти по дальнему адресу return(mem);
}
void far memfree(char far *тетогу)
farfree(memory); // функция Си для освобождения памяти по
// дальнему адресу
}
```

```
void main(void)
{
long int siz;
char *fname;
// Вызов загрузчика для начальной загруаки стандартного
// драйвера DIGPAK
if ( !LoadDigPak("SOUNDRV.COM") )
{
printf("Failed to load sound driver.\n");
exit(l);
} if ( !InitDigPak() ) // Инициализация драйвера DIGPAK
{
// Выгрузка драйвера из памяти в случае неудачной
// инициализации и завершение работы программы.
UnLoadDigPak();
printf("Failed to initialize sound driver.\n") ;
exit(l);
}
if ( LoadMidPak('MIDPAK.COM", "MIDPAK.ADV", "MIDPAK.AD") ) // Загрузка компонентов MIDPAK
{
printf("Loaded MIDPAK.COM MIDPAK.ADV and MIDPAK.AD into Low Men\n");
if ( InitMidPak() )
{
printf("MIDPAK driver initialized.\n");
fname = floadlow("TEST.XMI",&siz); // Загрузка музыки
if ( fname }
{
printf("Loaded TEST.XMI %d bytes long.\n",siz);
RegisterXmidi(fname,siz); // Регистрация XMIDI
// последовательности
printf("Sequence registered, now playing.\n") ;
PlaySequence(0); // Исполнить первую
// последовательность
SegueSequence (1,-1); // Зарегистрировать вторую
// последовательность на
// исполнение сразу же по
// достижении первого
// Контроллера 119
}
} else
printf("Failed to initialize MIDPAK driver.\n");
}
TestDigPak(); // Тестирование/демонстрация возможностей функций
// DIGPAK
UnLoadMidPak(); // Выгрузка MIDPAK из памяти,
// освобождение аппаратуры
UnLoadDigPak(); // Выгрузка DIGPAK из памяти,
// освобождение аппаратуры
RemoveVectorLoader() ;
}
void TestDigPak(void)
{
int i,key,sound;
printf("Loading digital sound effects.\n");
if ( LoadSounds() ) // Загрузка звуковых эффектов в память
{
// Создание меню звуковых эффектов
printf("Select an sound effect to play. [ESC] when finished playing around.\n");
for (i=0; i<NOBJ; i++)
{
printf("%c %s\n",i+'A',Names[i]) ;
}
do
{
if ( keystat() ) // если клавиша нажата
{
key = getkey(); // получить нажатую клавишу
```

```
if ( key >= ' a' && key <= ' z') key-=32;
// преобразовать нижний регистр к верхнему
if ( key >= 'A' && key <= 'Z')
{
sound = key-'A';
if ( sound < NOBJ ) PlaySound(sound);
}
} while ( key != 27 );
UnloadSounds(} ;
}
}
// загрузить все звуковые эффекты в память
int LoadSounds(void)
{
int fph;
long int siz.end;
int i, handler;
int select;
for (i=0; i<NOBJ; i++)
{
Sounds[i] = fload(Names[i], &siz);;
if ( !Sounds[i] )
{
printf("File '%s' not found.\n"/Names[i]) ;
return(0) ;
}
ssize[i] = siz;
snd.frequency = 11000;
snci. sound = Sounds [i];
snd.sndlen = ssize[i]; // задает длину звукового эффекта
//в байтах
MassageAudio(&snd) ;
printf("Sound Loaded '%s'.\n",Names[i]);
}
return(1) ;
}
void UnloadSounds(void) {
int i;
for (i=0; I < NALLOC; i++) memfree(Sounds[i]);
if ( soundbuffer )
{
realfree(snd) ;
realfree(soundbuffer);
}
NALLOC=0 ;
}
void PlaySound(int sound)
{
if ( !soundbuffer }
{
snd = (SNDSTRUC *) realalloc(sizeof(SNDSTRUC));
snd->frequency = 11000;
soundbuffer = realalloc(65535);
}
StopSound(); // ожидать до окончания предыдущего звукового эффекта
snd.frequency = 11000;
snd.sound = Sounds[i];
snd.sndlen = ssize[i]; // задает длину звукового эффекта
//в байтах
DigPlay2(&snd); // воспроизвести эффект
//в аппаратно-зависимом формате
}
```

АЛГОРИТМЫ, СТРУКТУРЫ ДАННЫХ И МЕТОДОЛОГИЯ ВИДЕОИГР

Пришло время поговорить о том, как же собрать воедино все, о чем рассказывалось в предыдущих главах этой книги, чтобы получить в результате настоящую компьютерную игру- Компьютерная игра —*программа синергистическая,* она не является просто совокупностью своих частей. Игра должна на время поместить игрока в другую реальность. Причем этот другой мир будет существовать только в компьютере и сознании игрока. Чтобы достигнуть этого мы должны понять, как зрительно и эмоционально воспринимает игру сам игрок, и на основании этого выработать подходящую концепцию архитектуры компьютерной игры. В этой главе мы затронем множеством, не всегда связанных между собой, но несомненно важных, если, конечно, вы все еще хотите превратить те смутные очертания, которые возникают в вашем воображении, в захватывающую компьютерную игру.

В этой главе много неочевидных выводов и намеков, поэтому читайте внимательно! Основные темы этой главы:

- Структуры данных для представления игрового пространства;
- Столкновения объектов;
- Представление игровых объектов;
- Структуры данных в компьютерных играх;
- Клонирование игровых объектов;
- Состояния объектов;
- Пользовательский интерфейс;
- Демонстрационные режимы;
- Сохранение состояния игры;
- Моделирование настоящего мира.

Структуры данных, используемые для представления игрового пространства

Если бы мы решили воссоздать некий мир во всех его подробностях, то эта глава заняла бы, наверное, миллион страниц. Поэтому в играх мы имеем дело с приближенной к действительности реальностью. Зачем нам создавать весь мир в нашей игре? Нам достаточно смоделировать только то, что имеет непосредственное отношение к сюжету.

Поскольку в качестве прообраза нашей игры мы выбрали Wolfenstein и DOOM, давайте поговорим о том, как создавался мир этих игр. А затем используем эту концепцию как фундамент для нашей собственной компьютерной игры. Но прежде чем сделать это, вкратце обсудим «стандартные» способы представления трехмерного мира на основе многоугольников.

В качестве примера возьмем гипотетические трехмерные скачки. В такой игре нам понадобятся деревья, дома, беговая дорожка, трибуны и пункт контроля. При стандартном способе представления пространства, состоящего из многоугольников, эта информация должна быть представлена на уровне объектов, то есть для каждого трехмерного объекта нам потребуется его модель. Эти модели будут представлены в виде списка многоугольников, заданных их вершинами. Затем каждому объекту надо будет сопоставить координаты, ориентацию и масштаб. Следовательно, если мы хотим, чтобы у нас было десять разных зданий, мы должны определить десять разных объектов: по одному на каждый тип зданий. Затем мы можем взять каждый объект и создать связанный список.

Завершив список, мы получим возможность строить (или *визуализировать)* каждый объект. Подобные же списки следует создать для структур данных деревьев, трибун и т. д.

Наконец, непрерывная беговая дорожка будет представлена списком прилегающих многоугольников. Таким образом, весь мир игры будет представлен в виде списков объектов. На рисунке 11.1 показана схема такого *обьектно-прогтуанственного представления* игрового пространства.

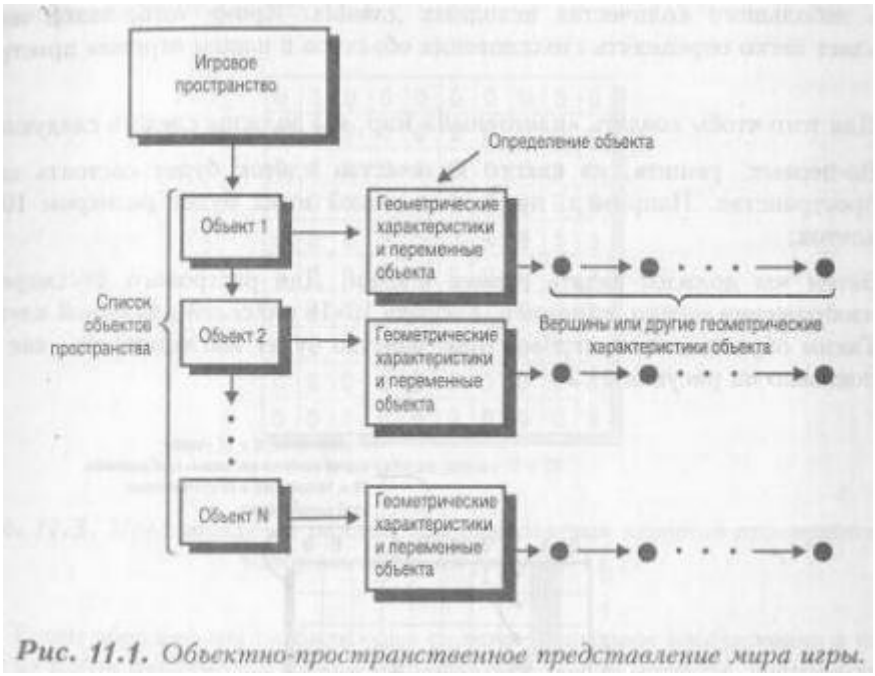


Рис. 11.1. *Объектно-пространственное представление мира игры.*

Это неплохой и вполне рабочий способ представления мира. Однако, представляя игровое пространство подобным образом, достаточно сложно определять столкновения объектов, создавать новые уровни и хранить данные. В компьютерных играх желательно иметь возможность быстро создавать новые Уровни на основе простых элементов. Поэтому стоит взглянуть на структуры Данных и методы их представления с точки зрения простоты создания нового варианта игрового пространства. Используя такой подход, мы можем проиграть в гибкости, но это окупится долгой жизнью игры.

Возьмем тот же пример скачек, реализуем его в виде двухмерного пространства, а затем придадим ему видимость трехмерности. В четырнадцатой главе, «Связь», мы подробно рассмотрим создание законченной компьютерной игры с «клеточным» пространством. Сейчас же просто попробуем разобраться в том, что собой представляет этот метод.

В большинстве компьютерных игр различные объекты и изображения многократно повторяются в разных комбинациях. Если вы внимательно посмотрите на игру РасМап, то заметите, что, несмотря на обилие лабиринтов, каждый из них состоит из одних и тех же объектов. Это таблетки, которые возвращают игроку силу, точки, повороты налево и направо, прямые стены и так далее, Пристальный анализ каждого экрана игры РасМап, покажет вам, что вся графика составлена из ряда «клеточек». Эти «клетки» в совокупности образуют повороты, прямые ходы и все остальное. Преимущество такого метода заключается в том, что он позволяет создать большой и разнообразный мир на основании очень небольшого количества исходных данных- Кроме того, такой метод позволяет легко определять столкновения объектов в нашем игровом пространстве.

Для того чтобы создать «клеточный» мир, мы должны сделать следующее:

- Во-первых, решить, из какого количества клеток будет состоять наше пространство. Например, пусть мир нашей игры будет размером 10x10 клеток;
- Затем мы должны задать размер клетки. Для растрового двухмерного изображения можно, например, выбрать 16x16 пикселей для одной клетки. Таким образом, наше игровое пространство будет выглядеть так, как это показано на рисунке 11.2.
- Затем мы должны нарисовать варианты клеток для нашей игры. Мы можем нарисовать клетки, содержащие изображения маленького дерева, стены, еды и так далее;
- После этого мы должны пронумеровать фрагменты. Таким образом, у каждого из них будет свой целочисленный идентификатор;
- Затем мы должны задать структуру данных (обычно в виде двухмерной матрицы) и заполнить ее соответствующими идентификаторами.

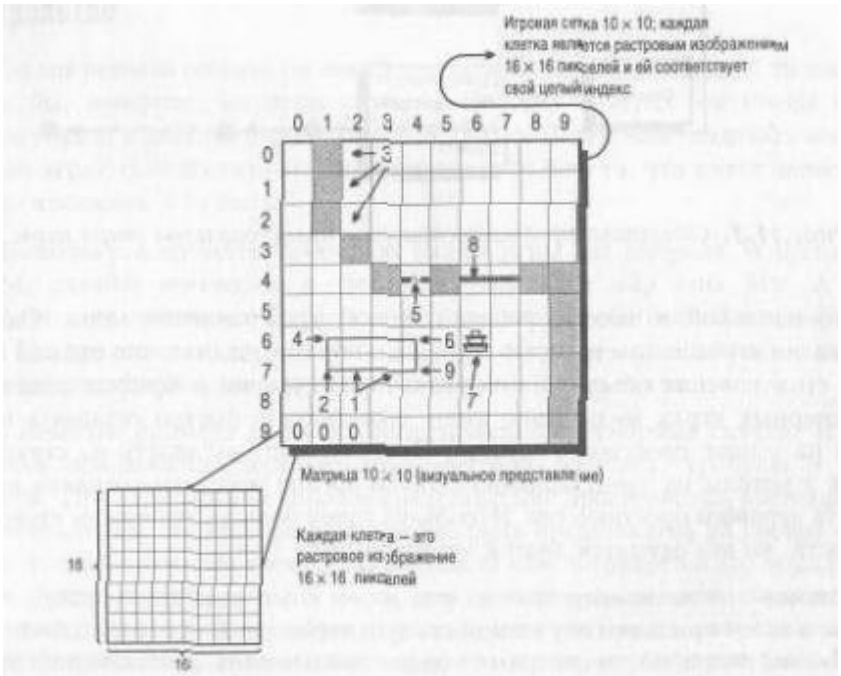


Рис. 11.2. *«Клеточное» представление игрового пространства.*

Преставление данных для нашего примера «клеточного» игрового пространства показано на рисунке 11.3.

Рис. 11.3. *Представление данных для определения игрового пространства, изображенного на рисунке 11.2.*

Таким образом, мы разбили одно сложное растровое изображение в нашей игре на набор фрагментов. Затем мы'создали игровую матрицу, которая задает растровое изображение как совокупность определенных фрагментов, используя Для их идентификации целые числа (идентификаторы фрагментов),

Теперь мы можем легко визуализировать игровое пространство: у нас есть его матрица, по которой несложно выбрать необходимые фрагменты изображения. Мы также имеем сами эти фрагменты (клетки). Фактически, мы просматриваем матрицу, клетка за клеткой, и в соответствии с идентификаторами воссоздаем необходимое растровое изображение.

На рисунке 11.4 показано построение растрового изображения «клеточного» игрового пространства.

Как вы можете видеть из рисунка 11.4:

- Мы заполнили структуру данных игры (обычно это двухмерная матрица) Целыми числами, причем каждое число - это идентификатор того фрагмента, изображение которого мы хотим поместить в соответствующей клетке;
- Затем во вложенном цикле определяем идентификаторы клеток пространства;
- В соответствии с каждым идентификатором визуализируем необходимый фрагмент растрового изображения. Несколько таких фрагментов будут составлять изображение здания, дороги, объекта игры и т. д.

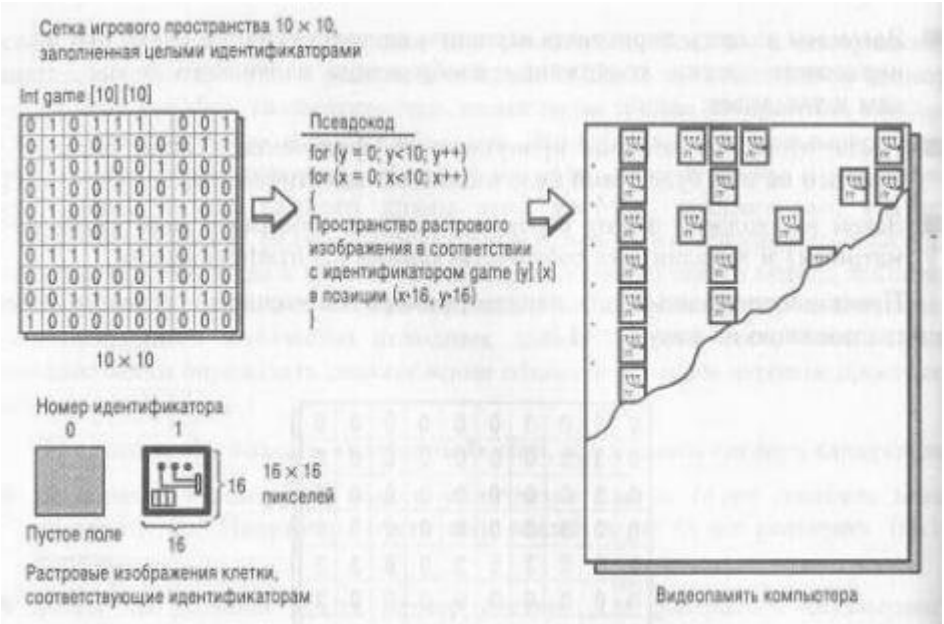


Рис. 11.4. *Построение растрового изображения «клеточного» игрового пространства.*

Мы должны только побеспокоиться о правильном позиционировании каждого фрагмента. В нашем примере при переходе к следующей клетке мы должныпередвигаться сразу на величину размера клетки, то есть на 16 пикселей.Нарисовав весь экран (10x10 клеток), мы получим растровое изображение160x160 пикселей.

Используя клеточное построение игрового пространства, легко создавать новые уровни в игре. Более того, такой метод упрощает создание инструментов для изготовления новых игровых пространств. Несложно написать программу, которая будет рисовать на экране сетку, а затем с помощью пиктограмм или цветов представлять то растровое изображение игрового пространства, которое вам нужно. То есть вы размещаете пиктограммы или помечаете цветом нужные клетки, а программа транслирует их в соответствующие идентификаторы фрагментов и записывает результирующую матрицу. Таким образом, в рекордно короткие сроки можно создать десятки различных уровней для своей игры, причем каждый из них займет минимум оперативной памяти.

Единственный недостаток данного метода заключается в том, что мы не можем поместить объект в произвольное место. Он обязательно должен вписываться в границы клетки. Однако обычно это не является проблемой.

Подобная техника была использована в DOOM и Wolfenstem. Только вместо фрагментов, представляющих плоское растровое изображение, там используются трехмерные кубы, составленные из одной или нескольких текстур. Создавая нашу игру, мы можем поступить так же.

Если вы играли в Wolfenstein, то наверняка заметили, что мир этой игры создан с - помощью набора кубов, которые покрыты текстурой с четырех вертикальных сторон. Каждый из этих кубов в действительности является клеткой в двухмерной решетке. Идентификаторы каждого модуля задают текстуры, которые будут расположены соответственно по четырем сторонам. Программа считывает эти значения и на их основании создает трехмерный мир.

Создать трехмерный мир с помощью двухмерной решетки помогает её регулярная структура и одинаковая высота всех кубов. Таким образом, подобное игровое пространство получается как бы в результате экструзии плоского клеточного мира.

Клеточное пространство удовлетворяет практически всем требованиям реальной игры. Однако иногда вам могут потребоваться некоторые дополнительные возможности, которые мы уже разбирали при обсуждении объектно-пространственного метода (метода многоугольников). С помощью клеточного подхода удобно создавать окружающую среду игры. Однако иногда ограничения этого метода делают невозможным создание определенных типов объектов, и тогда целесообразно использовать другую структуру данных. Например, иная структура просто необходима, если по ходу игры требуется размещать мелкие объекты в произвольных местах.

В трехмерном мире DOOM также имеются двухмерные стационарные объекты, такие как еда, оружие и т. д. Я подозреваю, что эти объекты имеют другую структуру данных, возможно связанную с конкретным помещением. Эта дополнительная структура данных задает определенное положение и тип объекта, который может быть расположен в данной комнате.

В любом случае, используйте те методы, которые позволяют реализовать ваши замыслы. Однако не забывайте о том, что новые уровни и новые объекты Должны создаваться как можно более простым путем. Вы не должны каждый Уровень строить с нуля. Вы должны иметь для этого инструменты и максимально использовать уже готовые элементы.

Теперь, создав игровое пространство, посмотрим как его объекты взаимодействуют между собой.

Когда объекты сталкиваются

В общем случае столкновение объектов происходит тогда, когда занимаемые ими пространства пересекаются (см. рис. 11.5).

Однако для игр на персональных компьютерах вычисление области пересечения произвольных изображений потребует слишком много времени. Поэтому большинство программистов используют в подобных расчетах *описывающие прямоугольники,* то есть минимальные прямоугольники, в которые помещается весь объект.

Контролировать столкновения между прямоугольниками намного проще. Например, пусть даны прямоугольники R1 и **R2.** Алгоритм 11.1 контролирует их столкновение.

Алгоритм 11.1. Контроль столкновений с помощью описанных прямоугольников.

```
For (для каждой вершины прямоугольника R1) do
if (проверить координаты X и У для вершин прямоугольника R2)
{
есть столкновение
выход
} } // конец
```

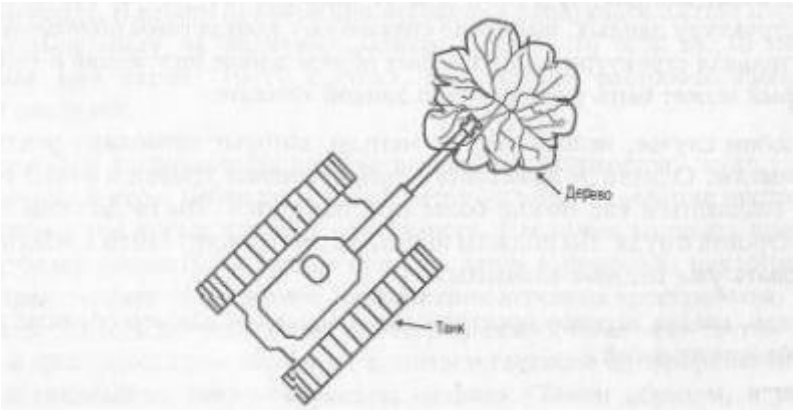


Рис. 11.5. Столкновение между двумя игровыми объектами.

Существуют более эффективные методы контроля столкновений прямоугольников, но основная идея, я думаю, понятна. Использовать этот метод не так уж и плохо. Однако может возникнуть такая ситуация, когда мы решим, что произошло столкновение объектов, а на самом деле пересеклись только описанные прямоугольники, а не истинные изображения. Рисунок 11.6 поясняет такую ситуацию.

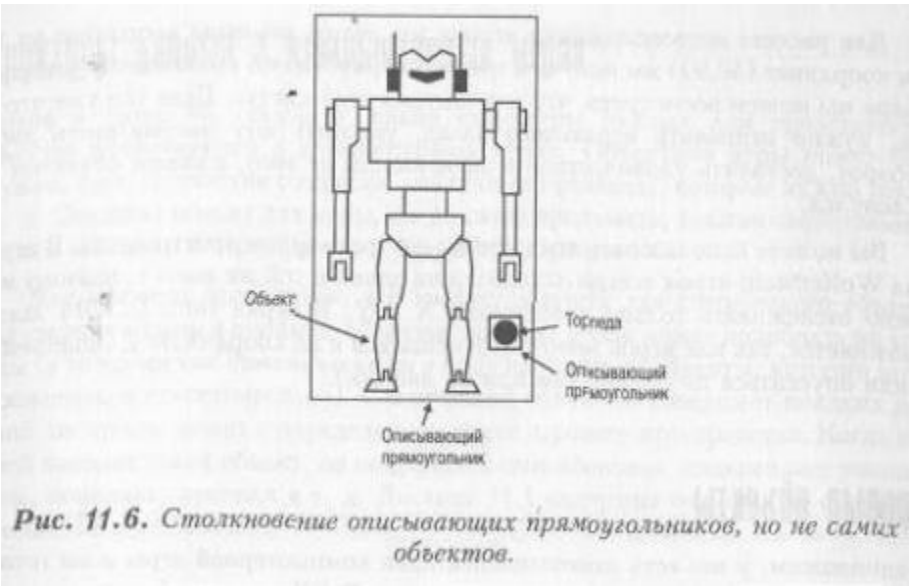


Рис. 11.6. Столкновение описывающих прямоугольников, но не самих объектов.

Чтобы, это исправить, мы можем либо уменьшить на несколько процентов размер ограничивающего объект прямоугольника, либо определять столкновения в два шага:
.

- Вначале для выявления возможного столкновения использовать описанные прямоугольники по Алгоритму 11.1;
- Затем делать более тонкую проверку вероятных столкновений, отобранных после первого теста, уже с учетом истинной геометрической формы объекта.

Я вам советую просто использовать ограничивающий прямоугольник, размер которого на десять процентов меньше исходного, и игрок будет доволен.

Мы уже обсуждали основные аспекты столкновений объектов между собой в четвертой главе, «Механизмы двухмерной графики». Теперь же обсудим столкновения со стенами и другими объектами, образующими само игровое пространство -обычно такие столкновения рассматривают как отдельную задачу.

Определение столкновений в «клеточном» пространстве тривиально. Все, что надо сделать, это определить, в какую клетку перемещается игрок и осмотреть, не занята ли она уже каким-либо объектом. Например, в рассмотренном выше примере пространство имело размер 10x10 клеток, а каждая клетка -16x16 пикселей. Общий размер изображения на экране составлял 160 пикселей. Если координаты игрока принимают значение (50,92), то он находится внутри какой-то клетки. Если же внутри этой клетки уже что-то есть, о наш игрок не должен в нее попадать!

Для расчета местоположения игрока мы делим значения координат на 16 Для координат (50,92) мы получим третью сверху, пятую слева клетку решетки Теперь мы можем посмотреть, что находится в этой клетке. Если там уже что-то есть, нужно отправить играющего назад, устроить ему неприятность либо наоборот, доставить удовольствие в зависимости от того, с каким объектом он столкнулся.

Вы можете использовать эту технику и в трехмерном пространстве. В играх типа Wolfenstein игрок всегда находится на одной и той же высоте, поэтому нам нужно отслеживать только координаты X и Y. В играх типа DOOM задача усложняется, так как игрок может перемещаться и по координате Z (подниматься или опускаться по ступенькам или на лифтах).

Игровые объекты

Предположим, у вас есть замечательная идея компьютерной игры и вы готовы написать 50000 или более строк программы на Си! Первая проблема, с которой вы столкнетесь при создании своей игры, будет связана с представлением объектов. Должны ли вы использовать массивы, структуры, связанные списки или что-то еще? Мой совет будет такой: «Начинайте с простого, мудрость приходит с опытом».

Мы будем использовать время от времени и массивы, и структуры, и связанные списки. Двоичные деревья поиска и другие экзотические структуры не стоит использовать, по крайней мере, до тех пор, пока вы не напишете хорошо работающие алгоритмы или, может быть, даже системы искусственного интеллекта. Используйте то, что у вас хорошо получается. Если вы можете сделать что-то более элегантное (например, вместо массивов использовать связанные списки) - делайте это, а если нет, то и не надо.

Старайтесь, чтобы выбранные вами структуры данных повышали эффективность работы программы. Например, пусть совершенно точно известно, что в вашей игре будет не более десяти объектов. Вы, конечно, можете создать связанный список и пополнять его по мере создания объектов. Это сэкономит память, но сделает программу более сложной. Столько хлопот ради десятка объектов - так ли уж это нужно? Простой массив в такой ситуации куда как уместнее!

Компьютерные игры настолько сложны, что если вы будете еще и сами усложнять простые вещи, то никогда не закончите работу. Выполните ее в черновом варианте, а затем, если хотите, возвращайтесь и совершенствуйте. У меня есть друг, который в 70-х или 80-х годах создал одну очень знаменитую компьютерную игру. Он всегда говорил мне: «Напиши игру, потом перепиши ее, а затем напиши еще лучше». Таким образом, начинайте с чернового варианта, а затем уже шлифуйте его.

Структуры данных в компьютерных играх

Теперь я хотел бы обсудить, какие структуры данных для представления объектов используются в компьютерных играх. Объектами игры может быть оружие, фантастические создания, различные предметы, которые нужно искать ц т. д. Создавая объект для игры, вы должны продумать, какими свойствами он должен обладать и отразить это в структуре данных.

Для примера рассмотрим, что требуется иметь для статического объекта. Под *статическими я* понимаю объекты, которые игрок может подобрать по ходу игры (в то время как *динамическими* я буду называть те объекты, которые могут перемещаться самостоятельно). Статический объект не совершает никаких действий, он просто лежит в определенном месте игрового пространства. Когда ваш герой находит такой объект, он поправляет свое здоровье, подкрепляет упавшие силы, пополняет арсенал и т. д. Листинг 11.1 содержит возможную структуру данных такого объекта.

Листинг 11.1. Структура данных статического объекта.

```
typedef struct static typ
{
int x,y; //позиция объекта
int type; // тип объекта: еда, энергия или здоровье
char *data; // указатель на растровое изображение объекта
int state; // состояние объекта
int amount; // поле для уточнения типа объекта
} static, *static ptr;
```

Для начала вполне можно использовать Листинг 11.1, а потом дополнять его необходимыми полями. Мы могли бы создать связанный список с использованием указателей, однако если объектов в игре немного, то лучше использовать массив, описав его следующим образом:

```
static stuff [10]
```

Осталось инициализировать массив stuff, а затем, работая со статическими объектами, ссылаться на него.

Другой пример. Давайте рассмотрим, чем обладает в игре сам игрок. В нашей демонстрационной игре Warlock игрок - это волшебник, он имеет заклинания, здоровье, несколько жизней, а также может собирать различные предметы. Кроме этого, нам понадобятся различные растровые изображения его рук, видимых на переднем плане экрана. Листинг 11.2 показывает первое приближение обходимой структуры данных.

Листинг 11.2. Структура данных игрока.

```
typedef struct player_typ
{
int x,y;                 // Позиция игрока
int lifes;               // Количество оставшихся жизней
int health;              // Уровень здоровья
int weapons[3];         // Массив, содержащий типы оружия, которые
// есть у игрока
int spells [10] ;        // Массив, содержащий заклинания, которые
// имеются у игрока
char *hands_stationary; // Растровое изображение рук игрока, когда
// он ничего не делает
char *hand_motion[4];   // Четыре растровых изображения рук игрока
// для выполнения заклинаний
int state;               // Состояние игрока: жив, мертв, умирает
} player, *player_ptr;
```

(Мы коротко расскажем о состояниях объектов чуть позже в этой главе, а более подробно —в тринадцатой главе, "Искусственный интеллект").

Как вы можете видеть, мы определили большинство необходимых переменных. Кое-что мы, конечно, добавим или выкинем позже, но для начала эта структура вполне подойдет. Это простой пример того, как следует создавать объекты в своей игре. Исходя из того, какая информация вам понадобится, организуйте соответствующую структуру и двигайтесь дальше.

Если в вашей игре есть как статические, так и динамические объекты, вы должны написать код, который будет рисовать, передвигать, отрабатывать столкновения и уничтожать объекты. Однако вам наверняка не захочется писать новые функции для каждого отдельного дерева! Давайте посмотрим, как можно клонировать игровые объекты.

Планирование игровых объектов

Под клонированием игровых объектов я понимаю повторное использование кода для множественной инициализации объектов одного типа, где каждый тип объекта имеет свою собственную копию данных. Рисунок 11.7 иллюстрирует это определение.

Для реализации клонирования каждый алгоритм (будь то рисование, движение или определение столкновения) пишется для работы с данными каждого отдельного экземпляра. Однако каждая функция пишется для циклической работы со всеми экземплярами объекта этого типа.

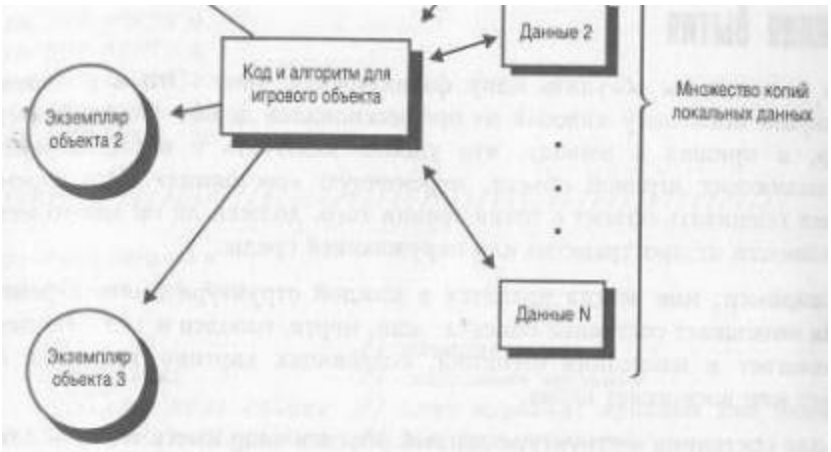


Рис. 11.7. Клонирование игровых объектов.

Например, у нас есть текст программы полета мухи в игровом пространстве и не более того. Мы можем использовать функции, приведенные в Листинге 11.3.

Листинг 11.3. Функции полета мухи.

```
void Erase_Flies(void) // удаление мух
for (каждой структуры данных полета мухи) do
{
удаление мухи
} // конец цикла
void Move_Flies(void) // перемещение мух
for (каждой структуры данных полета мухи) do
```

```
{
current_fly.x+=current_fly.xv; // перемещение по оси X
current_fly.y+=current_fly.yv; // перемещение по оси У
}
```

Мы могли бы создать похожие функции для рисования мух и контроля за их столкновениями. Однако я думаю, что вы уловили суть. Мы создаем одну процедуру для каждого действия над объектом и затем создаем множество объектов данных. Например, если бы у нас было тысяча мух, каждая из функций работала бы с тысячью структур.

Состояния бытия

Теперь я хотел бы обсудить одну философскую тему. Это и в самом деле философия, поскольку каждый из профессионалов делает это по-своему. Например, я пришел к выводу, что удобно включить в набор переменных, представляющих игровой объект, переменную «состояния». Эта переменная помогает оценивать объект с точки зрения того, должен ли он как-то меняться в зависимости от пространства или окружающей среды.

Например, мне всегда нравится в каждой структуре иметь переменную, которая описывает состояние объекта: жив, мертв, голоден и т, д. Эта переменная помогает в написании функций, создающих картину того, как объект погибает или воскресает вновь.

Поле состояния в структуре данных объекта надо иметь также и для того, чтобы использовать некоторые функции, изображающие объект в тот момент, когда он совсем здоров и когда он умирает. Эта особая информация берется из переменных состояния. Более подробно мы рассмотрим это в тринадцатой главе, «Искусственный интеллект». Однако сейчас нам надо знать, что система или объект в компьютерной игре (как и в реальной жизни) может проходить ряд состояний. Для того чтобы этого достичь, мы будем использовать переменные состояния в структуре данных каждого объекта.

Как пример использования состояния для оценки некоторых аспектов объекта игры давайте посмотрим на маленькую программу, создающую муравья. Муравей может ходить в разные стороны, пока не наткнется на стену или камень, после чего он выбирает новое направление (на север, юг, запад или восток). Так как четыре направления отслеживаются достаточно просто, мы можем воздержаться от использования переменных состояния. Однако движение на север, юг, запад и восток могут быть легко заменены такими эмоциями как чувство голода, усталость, счастье или гнев.

Для наглядности, используя наш метод имитации игровых объектов, я создал целый муравейник. Листинг 11.4 показывает эту программу.

Листинг 11.4. Муравейник (ANTS.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ////////////////////////////////////////
#include <dos.h>
#include <bios.h>
#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <graph.h>
// ОПРЕДЕЛЕНИЯ /////////////////////////////////////////////
#define ANT_NORTH 0
#define ANT_EAST 1 #define ANT_SOUTH 2
#define ANT_WEST 3 #define NUM_ANTS 50
// СТРУКТУРЫ ///////////////////////////
// структура муравья
typedef struct ant_typ
{
int x,y;              // позиция муравья
int state;            // состояние муравья
unsigned char color; // цвет муравья: красный или зеленый
unsigned back_color; // фон под муравьем
} ant, *ant_ptr;
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ ///////////////////////////////////
unsigned char far *video_buffer: = (char far *)0xA0000000L;
// указатель на видеобуфер
unsigned int far *clock = (unsigned int far *)0x0000046C;
// указатель на внутренний таймер
// наши маленькие муравьи
ant ants[NUM_ANTS];
// ФУНКЦИИ ////////////////////////////////////////////////
void Timer(int clicks)
{
// эта функция использует внутренний таймер с частотой 18.2 "тик"/с.
//32-битовое значение этого таймера находится по адресу 0000:046Ch
```

```
unsigned int now;
//получаем текущее время
now = *clock;
//Ожидаем до истечения указанного периода времени.
//Заметьте; что каждый,"тик" имеет длительность примерно в 55мс.
while(abs(*clock - now) < clicks){}
) // конец Timer
////////////////////////////////////////////////////////
void Plot_Pixel_Fast(int x,int y,unsigned char color)
{
// эта функция рисует точку заданного цвета несколько быстрее чем
// обычно, за счет применения операции сдвига вместо операции
// умножения
// используется тот факт, что 320*y = 256*y + 64*y = y<<8 + y<<6
video_buffer[ ( (y<<8) + (y<<6) ) + x] = color;
} // конец Plot_Pixel_Fast ////////////////////////////////////////////////////////////
unsigned char Read_Pixel_Fast(int x,int y)
{
// читаем значение пикселя из видеобуфера
return (video_buffer [((y<<8) + (y<<6) ) + x]);
} // конец Read_Pixel_Fast
//////////////////////////////////////
void Draw_Ground(void)
{
int index;
// эта функция рисует разбросанные по экрану серые камешки
for (index=0; index<200; index++)
{
Plot_Pixel_Fast(rand()%320,rand()%200, 7 + rand()%2);
} // конец цикла
} // конец Draw_Ground ////////////////////////////////////////////
void Initialize_Ants(void)
{
int index;
for (index=0; index<NUM_ANTS; index++)
{
// выбираем случайным образом начальную позицию, цвет и состояние
// для каждого муравья, а также определяем его фон
ants[index].x     = rand(}%320;
ants[index].y     = rand()%200;
ants[index].state = rand()%4;
if (rand()%2==1)
ants[index].color = 10;
else
ants[index].color = 12;
// сканирование фона
ants[index].back_color = Read_Pixel_Fast(ants[index].x, ants[index].y);
} // конец цикла
} // конец Initialize_Ants ////////////////////////////////////////////////////////////
void Erase_Ants(void)
{
int index;
// в цикле обрабатывается массив муравьев, все муравьи замещаются
// точками, цвета соответствующего фона
for (index=0; index<NUM_ANTS; index++)
{
Plot_Pixel_Fast(ants[index].x, ants[index].y,
ants[index].back_color) ;
} // конец цикла
} // конец Erase Ants
////////////////////////////////////////////////////////
void Move_Ants(void)
{
int index,rock;
//в цикле обрабатывается массив муравьев, каждый муравей перемещается
//в соответствии со своим состоянием
for (index=0; index<NUM ANTS; index++)
```

```
{
// каково состояние муравья?
switch(ants[index].state)
{
case ANT_NORTH:
{
ants[index].y—;
} break;
case ANT_SOUTH:
{
ants[index].y++;
} break;
case ANT_WEST:
{
ants[index].x--;
} break;
case ANT_EAST:
{
ants[index].x++;
} break;
} // конец оператора switch
// проверка, не столкнулся ли муравей
// с границами экрана или с камнем
if (ants[index].x > 319) ants[index].x = 0;
else if (ants[index].x <0)
ants[index].x = 319;
if (ants[index].y > 200)
ants[index].y = 200;
else if (ants[index].y <0)
ants[index].y = 199;
// здесь проверяем, не столкнулся ли муравей с камнем
rock = Read_Pixel_Fast(ants[index].x, ants[index].y);
if (rock)
{
// изменение состояния
ants[index].state =rand()%4;  // выбор нового состояния
} // конец оператора if
} // конец цикла
} // конец Move_Ants
////////////////////////////////////////////
void Behind_Ants(veid)
{ int index;
// в цикле обрабатывается массив муравьев,
// определяется цвет фона для каждого муравья
for (index=0; index<NUM_ANTS; index++)
{ // читается пиксель и его значение сохраняется
// для дальнейшего использования
ants[index].back_color = Read_Pixel_Fast(ants[index].x, ants[index].y) ;
} // конец цикла
} // конец Behind_Ants ////////////////////////////////////////////////////////
void Draw_Ants(void)
{
int index;
// в цикле обрабатывается массив муравьев, рисуется каждый
// муравей соответствующим цветом
for (index=0;index<NUM_ANTS; index++)
{
Plot_Pixel_Fast(ants[index].x, ants[index].y, ants[index].color);
} // конец цикла
} // конец Draw_Ants
// ОСНОВНАЯ ПРОГРАММА ////////////////////////////////////
void main (void)
{
// установка видеорежима 320x200x256
_setvideomode(_MRES256COLOR);
_settextposition(2,0);
printf("Hit any key to exit.");
```

```
//построение игрового пространства
Draw_Ground() ;
/ создаем муравьев
Initialize_Ants();
while(!kbhit())
{ // стираем всех муравьев
Erase_Ants (};
// перемещаем всех муравьев
Move_Ants ();
// определяем фон под муравьями
Behind_Ants() ;
// рисуем всех муравьев
Draw_Ants (),;
// немного подождем
Timer(2);
} // конец оператора while
// восстановление первоначального видеорежима
_setvideomode(_DEFAULTMODE);
} // конец функции main
```

Если вы запустите программу, то увидите кучу точек на экране. Красные и зеленые точки — это муравьи, а серые — это камни. Когда муравьи натыкаются на камни, они меняют направление движения. Эта программа показывает использование переменных состояния. Позже мы узнаем, как использовать переменные состояния для создания более сложных для вычисления структур, называемых *состоянием машины*.

Теперь изменим тему и поговорим о том, как выглядит игра с точки зрения игрока.

Интерфейс пользователя

Человек познает мир игры через графику и звук. Тем не менее, нельзя забывать еще об одной детали, влияющей на восприятие игры - интерфейсе пользователя. С помощью интерфейса пользователя игрок выбирает уровень сложности, настройки, задает количество участников и т. д. Поэтому интерфейс пользователя должен быть максимально простым в управлении. Ничто так не раздражает, как непонятные и запутанные команды и настройки. (Зачастую этим грешат спортивные игры. У них миллион настроек, и пока вы разбираетесь с ними, у вас пропадает желание поиграть).

В компьютерных играх с помощью интерфейса решаются две .задачи:

- Во-первых, интерфейс пользователя в течение всего времени игры выводит на экран жизненно важную информацию о статусе игры, количестве набранных очков и т. д.
- Во-вторых, с помощью интерфейса пользователя можно при необходимости изменять настройки игры.

Причем это должно осуществляться просто и понятно. Необходимо, чтобы игровой интерфейс акцентировал внимание на наиболее важных состояниях, возникающих в процессе игры. Более того, по-возможности надо использовать текст. Конечно, пиктограммы выглядят заманчиво, но как-то я столкнулся с пиктограммой, изображающей вулкан, но так и не понял, зачем она нужна.

Интерфейс пользователя должен удовлетворять следующим требованиям:

- Быть доступным в любой момент игры;
- Использовать слова, написанные большими буквами и не выводить на экран много пунктов настроек одновременно;
- С одной стороны позволять игроку настраивать программу, а с другой - защищать данные от уничтожения и случайных ошибок;
- Учитывать возможность того, что игрок не знает, что делает.

Интерфейс пользователя подобен сахарной глазури на кексе. Вы должны думать о том, как лучше всего ее употребить. Помните: интерфейс должен быть простым. Не загружайте его деталями и избегайте символов (если только они не будут абсолютно понятными).

Демонстрационный режим

После того как вы окончите работу над созданием компьютерной игры, вы наверняка захотите включить в нее режим демонстрации для показа ее возможностей. Если вы создавали игру с учетом такой необходимости (например, предусмотрели возможность получения исходных данных из файла, а не от игрока), то сделать это достаточно просто.

Во время игры изменяется состояние устройства ввода-вывода. (Более подробно об этом можно прочесть в третьей главе, «Основы устройств ввода»). С помощью этих устройств игрок может влиять на события. Во время игры логический контроль за вводом осуществляется с помощью определенной функции, такой, например, как Get_Input () . Эта функция отвечает за отслеживание текущего состояния устройства ввода.

Таким образом, все что мы должны сделать, это добавить в функцию контродя за вводом возможность считывания данных из заранее созданного файла. Сама по себе игра не почувствует никакой разницы. Рисунок 11.8 наглядцо показывает реализацию этого метода.

Рис. 11.8. *Использование файла в качестве источника исходных данных для реализации демонстрационного режима.*

Но, спросите вы, как создать такой файл? Для этого необходим особый режим игры. В таком режиме данные от устройства ввода будут оцифровываться с некоторым постоянным интервалом времени (скажем, 10 раз в секунду) и записываться в специальный файл. В демонстрационном режиме функция ввода будет вместо информации от устройства ввода использовать информацию из этого файла.

Это самый распространенный метод реализации демонстрационных режимов. Конечно, существуют и другие способы, но все они используют либо заранее подготовленные данные определенного формата, либо подобия системы искусственного интеллекта.

Вы можете решить, что сохранение всей информации о вводе потребует много памяти. Давайте посмотрим. Если мы сохраняем состояние устройства ввода 10 раз в секунду, то за минуту набежит 600 байт, а за 10 минут - всего 6 килобайт. Как видите, это не проблема, поэтому я советую вам пользоваться именно таким методом. Однако еще *до начала* составления программы вы должны предусмотреть в функции ввода «переключатель», с помощью которого можно выбирать между мышью, клавиатурой, джойстиком или воспроизведением файла. Возможность воспроизведения файла позволяет также реализовать в игре режим "замедленного повтора".

Сохранение игры

Физики верят, что если бы они знали координаты и состояние всех частиц Вселенной, то смогли бы предсказывать будущее. Возможно, это и так. Перед нами же стоит задача попроще. Нам надо уметь записывать состояние нашей игры с тем, чтобы при необходимости его можно было восстановить.

Итак, проблема заключается в следующем: как запомнить игру так, чтобы в любой момент мы могли восстановить ее в прерванном месте? Ответ довольно Прост: надо создать список всех игровых объектов и факторов, которые могут измениться в течение игры. Затем создать файл определенного формата и записать в него всю эту информацию. Теперь, когда игрок хочет сохранить игру, мы просто записываем необходимую информацию на диск.

Для примера рассмотрим, как можно сохранить игру в DOOM или Wolfenstein 3-D:

- В каждой игре присутствует множество неподвижных объектов. Поэтому их сохранять не надо. Однако во время сохранения игры нам необходимо помнить, что они существуют;
- Также мы должны принимать во внимание настройки игры. Для каждой настройки мы должны запомнить копию ее структуры. Эти структуры должны содержать положение, состояние и т. д.;
- Наконец, надо запомнить, какой инвентарь имеет при себе играющий на данный момент, а также каково его состояние (позиция, здоровье и т. д.), иными словами, то, что необходимо для правильного сохранения статуса игры.

Это все. Если кратко, мы делаем моментальный снимок нашего мира и представляем его в оцифрованном виде.

Моделирование реального мира

Хоть в нашей игре реальный мир будет, в основном, составлен из стен, тем не менее, мы должны хотя бы вкратце обсудить эту тему. Конечно, вам захочется, чтобы в вашей игре были те эффекты, с которыми вы сталкиваетесь в реальном мире и которые вызываются силами трения, притяжения, сохранения импульса и т. д. Например, без них не обойтись при имитации скачек, при запуске ракеты в военных играх и при столкновении шаров в симуляторе бильярда. В любом лучае, я хочу вам преподать маленький курс основ механики.

Каждый из нас знает, что такое скорость. Это просто изменение расстояния за единицу времени. Например, если я хочу перемещать объект со скоростью два пикселя в секунду, я могу это

сделать так:

```
Object_Position += 2;
```

Если вы посмотрите на координаты объекта как на функцию времени, та увидите график, приведенный на рисунке 11.9.

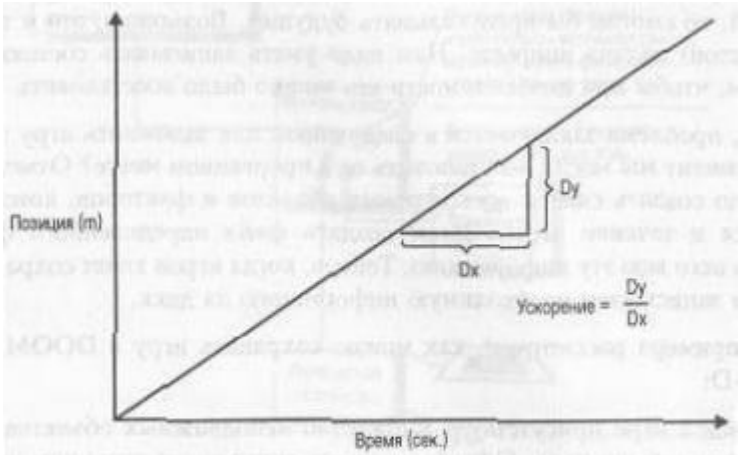


Рис. 11.9. *График скорости, как функции времени.*

Что такое ускорение? Ускорение - это изменение скорости за единицу времени или, другими словами, приращение скорости. Когда вы в своем автомобиле нажимаете на акселератор, то скорость становится все 6ольше ибольше. Ускорение может быть записано в следующем виде:

```
Object_Position += velocity;
velocity += acceleration_factor;
```

Приведенный фрагмент программы увеличивает скорость. Это выглядит как ускорение. Чаще всего мы сталкиваемся с ускорением свободного падения. Когда объект падает на землю, его ускорение составляет 9,8 м/сек . То есть если в первую секунду падения его скорость достигнет 9,8 м/сек, то во вторую она вырастет до 19,6 м/сек.

Чтобы увидеть, как это осуществляется, посмотрим на текст программы в Листинге 11.5. Она показывает падение мяча под действием силы тяжести. Вы можете изменять ускорение с помощью клавиш + и -.

Листинг 11.5. Падение мяча (BALL.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ////////////////////////////////
#include <stdio.h>
#include <math.h>
#include <graph.h>
//ОПРЕДЕЛЕНИЯ //////////////////////////
#define EARTH_GRAVITY 9.8
//ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ ////////////////////////////
unsigned int far *clock = (unsigned int far *)0x0000046C;
                        // указатель на внутренний таймер
// ФУНКЦИИ ////////////////////////////////
void Timer(int clicks)
{
// эта функция использует внутренний таймер с частотой 18.2 "тик"/с.
// 32-битовое значение этого таймера находится по адресу 0000:046Ch
unsigned int now;
// получаем текущее время now = *clock;
// ожидаем до истечения указанного периода времени. Заметьте, что
// каждый "тик" имеет длительность примерно в 55мс.
while(abs(*clock - now) < clicks){}
} // конец Timer
// ОСНОВНАЯ ПРОГРАММА ////////////////////////////////
void main(void)
{
float ball_x   = 160, ball_y = 50, ball_yv = 0, ball_acc = EARTH_GRAVITY;
int done=0, key;
// используем графические функции Си
_setvideomode(_MRES256COLOR) ;
_settextposition(0,0);
printf("Q to quit, use +,- to change gravity.");
while(!done)
{ // была ли нажата клавиша?
if (kbhit())
```

```
{
// какая клавиша была нажата?
switch(getch())
{
case ' -' :
{
ball_acc-=.1;
} break;
case '=':
{
ball_acc+=.1;
} break;
case 'q' :
{
done=1;
} break;
} // конец оператора switch
// сообщим игроку новое значение ускорения
_settextposition(24,2);
printf("Gravitational Constant = %f",ball_acc);
} // конец оператора if
// стираем изображение мяча
_setcolor(0) ;
_ellipse(_GBORDER, ball_x,ball_y,ball_x+10,ball_y+10);
// перемещаем мяч
ball_y+=ball_yv;
// увеличиваем ускорение
ball_yv+=(ball_acc*.1);
// проверим, не достиг ли мяч пола
if (ball_y>190)
{
ball_y=50;
ball_yv=0 ;
} // конец оператора if
// рисуем мяч
_setcolor(l) ;
_ellipse(_GBORDER, ball_x,ball_y,ball_x+10,ball_y+10) ;
// немного подождем
Timer(2);
} // конец while
// восстановить начальный видеорежим
_setvideomode(_DEFAULTMODE);
} // конец функции main
```

Физические модели нужны для того чтобы все происходящее в игре выглядело более реально и, кроме того, для того чтобы представлять себе, как наш объект будет взаимодействовать со статическими объектами.

В качестве последнего примера рассмотрим упругое столкновение. Программа текст которой приведен в Листинге 11.6, создает группу атомов, сталкивающихся в замкнутом пространстве. Когда атомы сталкиваются со стенкой резервуара, они отскакивают, сохраняя при этом свою кинетическую энергию. Мы могли бы использовать систему физических уравнений, однако нам надо только, чтобы это хорошо выглядело. Если вы посмотрите на столкновение бильярдных шаров с портиком стола, то увидите, что они всегда отскакивают под тем же углом, под которым ударились (то есть их угол падения равен углу отражения), как это показано на рисунке 11.10.

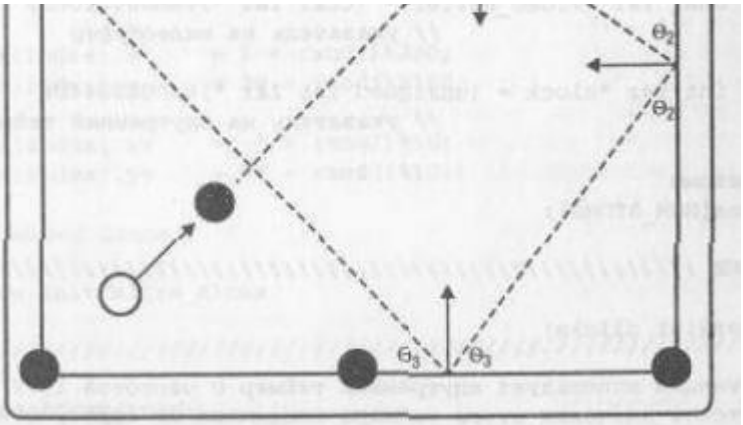


Рис. 11.10. *Траектория ударов.*

Таким образом, чтобы смоделировать это, мы должны только отразить значение скорости атома и все будет выглядеть «корректно с точки зрения физики». Текст программы приведен в Листинге 11.6.

Листинг 11.6. Идеальный газ (GAS.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ////////////////////////////////////////
#include <dos.h>
#include <bios.h>
#include <stdio.h>
#inciude <math.h>
#include <conio.h>
#include <graph.h>
// ОПРЕДЕЛЕНИЯ /////////////////////////////////////////
#define NUM_ATOMS 300
// СТРУКТУРЫ ///////////////////////////////////////////////
// структура атома
typedef struct ant_typ
{
int x,y;            // позиция атома
int xv,yv;        // скорость атома
} atom, *atom_ptr;
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ' ////////////////////////////////////
unsigned char far *video_buffer = (char far *)0xA0000000L;
// указатель на видеобуфер
unsigned int far *clock = (unsigned int far *)0x0000046C;.
// указатель на внутренний таймер
// наши атомы
atom atoms[NUM_ATOMS];
// ФУНКЦИИ ////////////////////////////////////////////////
void Timer(int clicks)
{
// эта функция использует внутренний таймер с частотой 18.2 "тик"/с.
// 32-битовое значение этого таймера находится по адресу 0000:046Ch
unsigned int now;
// получить текущее время
now = *clock;
// Ожидаем до истечения указанного периода времени. Заметьте, что
// каждый "тик" имеет длительность примерно в 55мс.
while(abs(*clock - now) < clicks){}
} // конец Timer
//////////////////////////////////////////////////////////
void Plot_Pixel_Fast(int x,int y,unsigned char color)
{ // эта функция рисует точку заданного цвета несколько быстрее чем
//обычно, за счет применения операции сдвига вместо операции // умножения
// используется тот факт, что 320*y = 256*y + 64*y = y<<8 + y<<6
video_buffer[((y<<8) + (y<<6)} + x] = color;
} // конец Plot_Pixel_Fast ////////////////////////////////////////////
void Initialize_Atoms (void)
{
int index;
for (index=0; index<NUM_ATOMS; index++)
{
// выбор случайного положения и траектории для каждого атома
//и их фона
atoms[index].x     = 5 + rand()%300;
atoms [index] .y     = 20 + rand()%160;
atoms [index] .xv    = -5 + rand() %10;
atoms[index].yv    = -5 + rand()%10;
} // конец цикла
} // конец Initialize_Atoms
////////////////////////////////////////////////
void Erase_Atoms(void)
{
int index;
// обрабатываем в цикле все атомы, стирая их изображения
for (index=0; index<NUM_ATOMS; index++)
```

```
{
Plot_Pixel_Fast( atoms[index].x, atoms[index].y, 0);
} // конец цикла
} // конец Erase_Atoms
//////////////////////////////////////////////////////
void Move_Atoms(void)
{ int index;
// обрабатываем в цикле все атомы, перемещая каждый атом.
// Также проверяем столкновение атомов со стенками контейнера.
for (index=0; index<NUM_ATOMS; index++)
{
// переместить атомы
atoms[index].x+=atoms[index].xv;
atoms[index].y+=atoms[index].yv;
// если атом столкнулся со стенкой, меняем знак скорости
// на противоположный
if (atoms[index].x > 310 11 atoms[index].x <10)
{
atoms[index].xv =-atoms[index].xv;
atoms [index].x+=atoms [index].xv;
} // конец оператора if
if (atoms[index].y > 190 11 atoms[index].y <30)
(
atoms[index].yv=-atoms[index].yv;
atoms[index].y+=atoms[index].yv;
} // конец оператора if
} // конец цикла
} // конец Move_Atoms
////////////////////////////////////
void Draw_Atoms(void)
{
int index;
// рисуем все атомы в цикле
for (index=0; index<NUM_ATOMS; index++)
{
Plot_Pixel_Fast (atoms[index] .x, atoms [index] .y, 10) ;
} // конец цикла
} // конец Draw_Atoms
// ОСНОВНАЯ ПРОГРАММА ////////////////////////////////////
void main(void)
{
// устанавливаем режим 320x200x256
_setvideomode(_MRES256COLOR) ;
_settextposition[2,0);
printf("Hit any key to exit.");
// рисуем контейнер _setcolor(9);
_rectangle(_GBORDER,0,16,319,199);
// инициализируем атомы
Initialize_Atoms();
while(!kbhit())
{
// стираем все атомы
Erase_Atoms () ;
// перемещаем все атомы
Move_Atoms();
// теперь рисуем атомы
Draw_Atoms();
// немного подождем
Timer(1);
} // конец while
// восстанавливаем исходный видеорежим
_setvideoroode(_DEFAULTMODE) ;
}// конец функции main
```

Все, физики с меня достаточно! Я думаю, что идея вам понятна: когда вы хотите что-то смоделировать, посидите и разберитесь с несколькими правилами или уравнениями, описывающими этот процесс. После того как вам все станет ясно, найдите самый простой путь воссоздания этого процесса в вашей игре.

ИТОГ

В этой главе был представлен потрясающий курс, посвященный идеям и способам представления мира игры и действиям в нем. Среди прочего мы узнали:

- Об объектно-пространственном и модульном методах представления мира игр;
- О демонстрационных режимах;
- О моделировании физических явлений.

Главная идея, которую вы должны были понять, одолев эту главу, такова: здесь не изложены догмы, а просто рассмотрены возможные варианты осуществления тех или иных вещей. Если вы знаете метод лучше - используйте его. Мы заложили фундамент, а все остальное в ваших руках.

Мнимое время, прерывания и многозадачность

Если вы когда-нибудь играли в компьютерные игры... впрочем, такое начало главы совершенно не подходит для этой книги! Короче говоря, вы наверняка подметили, что все в хороших играх выглядит так, как будто происходит одновременно. Но я уверен, что в 99,9 процентах случаев это не так! Компьютер просто-напросто делает все настолько быстро, что создается впечатление многозадачности. Эта глава как раз и посвящена примитивным возможностям персонального компьютера по поддержке многозадачности. Кроме того, мы научимся писать «цикл игры», который, собственно, и создаетвпечатление одновременности событий игры.Мы также обсудим некоторые вопросы архитектуры компьютерных игр, относящиеся кразработке и написанию «самодостаточных» функций. В этой главе будут рассмотрены следующие вопросы:

- Восприятие игры;
- Многозадачность;
- Реализация многозадачности при помощи прерываний;
- Создание обработчиков прерываний на Си;
- Цикл игры;
- Автономные функции;
- Функции ответа;
- Перепрограммирование системных часов;
- Объединение компонентов многозадачности в единое целое;
- Пример обработчика прерываний № 1 - Там полно звезд...
- Пример обработчика прерывания № 2 - Ловим нажатия клавиш!

Восприятие игры

Человеку, который собрался поиграть в компьютерную игру, хочется чтобы она была интерактивной. В данном случае под словом *интерактивная* я подразу меваю то, что графическое изображение должно быть четким и быстро сменяться. При этом музыка должна звучать в соответствующем темпе и игра обязана мгновенно реагировать на действия игрока (по крайней мере, с точки зрения играющего). Игроку должно казаться, что все (музыка, графика, звуковые эффекты и т. д.) происходит одновременно. Теперь взглянем на это с точки зрения программиста.

Вообще-то, сделать так, чтобы разные события в игре происходили одновременно, сложно. Персональный компьютер — это не многозадачная система (во всяком случае, не для игр, работающих под управлением DOS). Более того, у персонального компьютера только один процессор. Следовательно, иллюзия реальности или *"реального времени"* должна создаваться как-то иначе. Этот иной способ опирается исключительно на скорость работы компьютера. Быстродействие компьютера настолько превышает скорость человеческого восприятия, что машина успевает выполнять все операции последовательно, а человеку кажется, что все происходит одновременно.

На самом деле, в компьютерной игре все происходит примерно следующим образом: мы получаем команды пользователя, реагируем на них в соответствии с логикой игры, выводим на экран изображения объектов и озвучиваем происходящие события, затем повторяем все снова и снова. Благодаря тому, что это происходит десятки, если не сотни, раз в секунду, нам удается заставить игрока поверить в существование созданного нами виртуального мира.

Что такое многозадачность?

Многозадачность — это не что иное, как одновременное выполнение нескольких процессов на одном и том же компьютере. Например, Microsoft Windows представляет собой многозадачную операционную систему (некоторые, возможно, возразят мне, что она еще нуждается в некоторых дополнениях для того, чтобы ее можно было назвать настоящей многозадачной системой, однако для нашего случая подобный пример вполне подойдет). При желании в Windows пользователь вслед за одной программой может запустить и другую. Компьютер при обработке данных разбивает каждую секунду на несколько «тиков». Каждый процесс или программа получает в свое распоряжение несколько таких «тиков» работы процессора. По истечении отведенного на эту программу времени процессор переходит к обработке следующей программы и так далее. Так продолжается до тех пор, пока не будут обслужены все выполняющиеся программы, после чего процесс повторяется. Компьютер работает настолько быстро, что пользователю кажется, будто все программы работают одновременно. На рисунке 12.1 показано, как реализуется многозадачность.

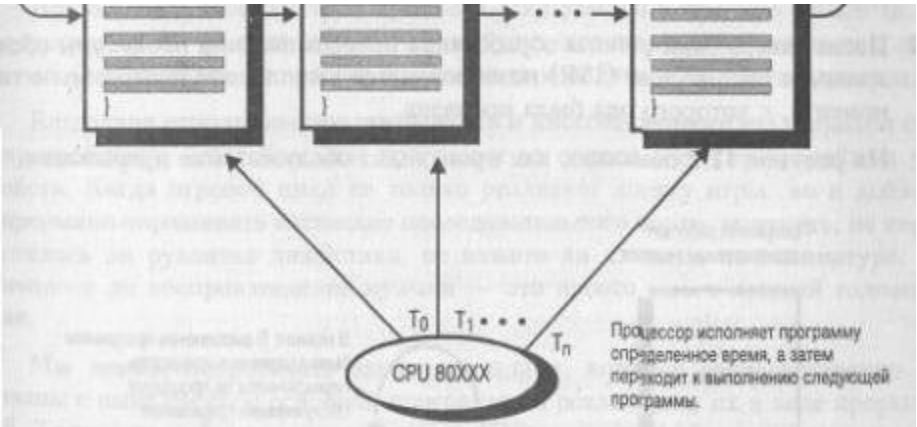


Рис. 12.1. *Пример многозадачности.*

При создании компьютерной игры нам незачем заботиться об обеспечении совместной работы нашей игры и, например, текстового процессора. Нас должна Интересовать организация совместной работы различных частей одной программы. Это означает, что нам нужно сделать так, чтобы каждая функциональная часть нашей игры обрабатывалась в течение некоторого времени, причем не слишком долго. (Под функциональной частью в данном случае я понимаю Фрагменты текста программы (подпрограммы), которые осуществляют вывод графического изображения, воспроизведение звука, а также реализуют логику самой игры). После того как каждая подпрограмма по очереди будет обработана, процесс должен повторится с начала, образуя *игровой цикл*.

Мы можем считать, что отдельная программа на Си, представляющая собой процедуру, вызывающую функциональные части, является своего рода маленьким виртуальным компьютером. При этом каждая из упомянутых функций это не что иное, как отдельная задача, выполняемая в псевдомногозадачном режиме. Отметим, что программа всегда контролирует выполнение функциональных частей, и игра может быть написана таким образом, что исполнение программы и функциональных частей никогда не прерывается другими процессами:

На самом деле это очень удобно: сложные компьютерные игры могут быть написаны и пишутся в виде единой программы безо всяких прерываний. Однако прерывания позволяют реализовать хоть и примитивную, но истинную многозадачность следующим образом;

- Выполнение программы может прерываться в любой произвольной точке. Состояние машины сохраняется таким образом, чтобы впоследствии можно было возобновить работу программы;
- После этого управление персональным компьютером передается обработчику прерывания, который представляет собой отдельную законченную программу;
- После завершения работы обработчика прерывания или процедуры обслуживания прерывания (ISR) возобновляется выполнение программы с того момента, с которого она была прервана.

На рисунке 12.2 показано, как происходит обслуживание прерывания.

Рис. 12.2. *Обслуживание прерывания.*

Вы можете сказать: «Но зачем нам это нужно?» Поверьте, есть много причин использовать прерывания. Ну, например, допустим, что некоторые данные вводятся в вашу программу с

клавиатуры. Возможен вариант, когда пользователь нажмет на клавишу в момент вывода па экран графического изображения. Тогда нажатие клавиши будет проигнорировано. Однако если при нажатии клавиши вызывается процедура обслуживания прерывания, то процессор переключит свою работу на обработку нажатия па клавишу. Таким образом можно быть уверенным, что информация о нажатии на клавишу никогда не будет потеряна.

В случае компьютерных игр нас интересует не столько ввод данных с клавиатуры, сколько *синхронизация.* Синхронизация для игры - это все. Изображение должно появляться и изменяться в нужный момент. Музыка должна звучать в подходящем темпе. Иногда мы хотим, чтобы то или иное событие произошло через определенный промежуток времени, скажем через 1/30 или 1/60 долю секунды. Без прерываний реализовать это на персональном компьютере сложно, так как понадобится отслеживать время» а это лишняя нагрузка на процессор.

Другая часто возникающая проблема заключается в том, что одна и та же подпрограмма на разных компьютерах будет исполняться с различной скоростью. Это означает, что расчет времени в игре является аппаратнозависимым.

Благодаря использованию прерываний и многозадачности мы решаем и еще одну проблему — освобождаем программу от опроса всех периферийных устройств. Когда игровой цикл не только реализует логику игры, но и должен непрерывно опрашивать состояние последовательного порта, выяснять, не переместилась ли рукоятка джойстика, не нажата ли клавиша на клавиатуре, не кончилось ли воспроизведение музыки — это просто много лишней головной боли.

Мы можем попробовать выделить задачи, которые непосредственно не связаны с выполнением основной программы и реализовать их в виде прерывании. Только не поймите меня превратно. Персональный компьютер не сможет предоставить нам в распоряжение тысячи прерываний на все случаи жизни. На самом деле мы сможем использовать лишь пару прерываний и только в определенных целях.

Если вы уже окончательно запутались, то... для чего же еще нужны друзья как не для того, чтобы помогать? Давайте попробуем вместе разобраться с этой путаной реализацией многозадачности в DOS.

Реализация многозадачности при помощи прерываний

При работе под операционной системой DOS, если и существует какой-нибудь способ, с помощью которого мы можем реализовать многозадачность, так это только использование механизма прерываний. Некоторые прерывания вырабатываются в результате внешних событий, в то время как другие евязаны с событиями внутренними. Для примера, давайте сначала рассмотрим, пожалуй самое популярное прерывание — прерывание от клавиатуры.

Всякий раз, когда происходит нажатие клавиши, ваша программа, что бы она при этом ни делала, останавливается и начинает работать процедура обслуживания прерываний клавиатуры. (Готов спорить, вы и не подозревали, что ваша программа останавливается при каждом нажатии на клавишу, однако это действительно так!) После окончания процедуры обслуживания прерывания управление снова передается вашей программе. Во время всего этого процесса ваша программа, данные и все остальное остается целым и невредимым. Для любой процедуры обслуживания прерываний это Правило Номер Один; без определенной цели ничего не должно уничтожаться. Так, например, если ваша процедура обслуживания прерывания использует для своей работы регистры процессора, вам первым делом следует сохранить содержимое этих регистров, затем осуществить обработку прерывания и снова восстановить содержимое регистров в точно таком же виде, каким оно было до прерывания.

Прежде чем мы начнем разбираться с основными принципами написания и установки обработчика прерываний, давайте взглянем, какие же прерывания есть у персонального компьютера. Посмотрите на таблицу 12.1.

Таблица 12.1. Прерывания ПК.

Номер	**Адрес**	**Функция**
0h	000-003h	Деление на ноль
1h	004-007h	Пошаговое выполнение
2h	008-00Bh	Немаскируемуе прерывание
3h	00C-00Fh	Точка останова
4h	010-013h	Переполнение
5h	014-017h	Печать содержимого экрана
6h	018-01Bh	Зарезервировано
7h	01C-01Fh	Зарезервировано
8h	020-	Таймер 18.2

	023h	
9h	024-027h	Клавиатура
0Ah	028-02Bh	Зарезервировано
0Bh	02C-02Fh	RS-232 Порт 1
0Ch	030-033h	RS-232 Порт 0
0Dh	034-03Bh	Жесткий диск
0Eh	038-03Bh	Дискета
0Fh	03C-03Fh	Зарезервировано
10h	040-043h	Функция видеовывода
11h	044-047h	Проверка оборудования
12H	048-04BH	Проверка памяти
13H	04C-04FH	Функции ввода/вывода на дискету
14H	050-053H	Функции ввода/вывода последовательного порта
15H	054-057H	Функции ввода/вывода на кассетный магнитофон
16H	058-05BH	Функции ввода клавиатуры
17H	05C-05FH	Функции вывода на принтер
18H	060-063H	Точка входа в ROM BIOS
19H	064-067H	Процесс загрузки
1AH	068-06BH	Получение информации о времени
1BH	06C-06FH	Управление прерыванием
1CH	070-073H	Управление таймером
1DH	074-077H	Таблица инициализации видеосистемы
1EH	078-07BH	Таблица параметров дискеты
1FH	07C-07FH	Таблица графических символов
20H	080-083H	Завершение DOS программы
21H	084-087H	Универсальные функции DOS
22H	088-08BH	Адрес завершения DOS
23H	08C-08FH	Адрес обработчика Ctrl+Break
24H	090-093H	Адрес обработчика критических ошибок DOS
25H	094-097H	Абсолютное чтение с диска DOS
26H	098-09BH	Абсолютная запись на диск DOS
27H	09C-09FH	Установка резидентной программы DOS
28-3FH	0A0-0FFH	Зарезервировано для DOS

40-7FH	100-1FFH	Не используется
80-F0H	200-3C3H	Зарезервировано для Бейсика
F1-FFH	3C4-3FFH	Не используется

Таблица 12.1 - это таблица векторов прерываний. Она занимает первые 1024 байт памяти каждого персонального компьютера. Всего в этой таблице 256 элементов, каждый из которых имеет размер 4 байта и представляет собой значение дальнего указателя на процедуру обслуживания прерывания. Как вы могли заметить, персональный компьютер не использует все 256 прерываний. Однако число задействованных прерываний постоянно растет.

Персональный компьютер поддерживает прерывания как на аппаратном, так и на программном уровне. Программные прерывания создаются с помощью расширенного набора инструкций процессора 80x86. Они были разработаны специально для того, чтобы дать возможность не только физическим устройствам мгновенно прерывать исполнение текущей программы. Большинство прерываний на персональном компьютере осуществляются программным путем. Однако некоторые осуществляются только с помощью аппаратуры (к ним относятся немаскируемые прерывания и прерывания от клавиатуры). С точки зрения программиста оба типа прерываний работают одинаково, поэтому нас это деление затрагивать не будет.

Внимание!

При использовании прерываний будьте очень осторожны: вы играете с огнем. Если вы допустите ошибку, компьютер может «зависнуть», что иногда приводит к потере важных данных. Будьте внимательны!

Итак, в соответствии с характером нашей игры мы должны выбрать нужные нам прерывания. Как их выбирать, еще не ясно, но мы разберемся с этим чуть позже. Затем нам нужно будет зарегистрировать (установить) свою собственную процедуру обработки прерываний. Вот, собственно, и все.

Единственное, чего нам не хватает для начала, это самой процедуры обработки прерываний, поэтому давайте разбираться, как она создается на языке Си.

Написание программы обработки прерываний на языке Си

Слава богу, мы можем написать эту программу на языке высокого, уровня. Даже страшно себе представить процесс создания процедуры обработки прерываний чисто на ассемблере. К счастью, современные расширенные компиляторы языка Си имеют специальные ключевые слова, позволяющие написать обработчик прерываний. Более того, Си берет на себя большую часть забот, связанных с организацией входных и выходных данных такой программы. Все что вы полжны сделать, это написать собственно программу обработки прерываний на стандартном Си. Чтобы сделать ее обработчиком прерывания, следует использовать ключевое слово interrupt.

Допустим, мы хотим создать обработчик прерывания, который планируем связать с прерыванием таймера - его номер 0x1C. Начать можно вот так:

```
void _interrupt _far Timer (void)
{
} // окончание функции timer
```

Ключевое слово _far необходимо потому, что все прерывания представляют собой 32-разрядные вызовы (смещение и сегмент).

В пределах функции вы можете делать все, что вам заблагорассудится с единственным условием: весь ваш код должен быть рентабельным. Это значит, что вы должны быть крайне осторожны при обращении к функциям DOS или сложным функциям Си типа printf. Я не рекомендую вам использовать в подобных функциях расширения языка Си. О сохранении содержимого регистров можно не беспокоиться: Си-компилятор запоминает содержимое регистров перед началом обработки прерывания и восстанавливает по окончании. Это касается всех регистров за исключением регистра сегмента стека SS. И еще, с целью облегчения доступа к данным при входе в обработчик прерывания регистр DS указывает на глобальный сегмент данных вашей программы, благодаря чему программа обработки прерываний имеет доступ к глобальным переменным. Звучит слишком заманчиво, чтобы быть правдой, однако на самом деле все организовано именно так.

Итак, у нас есть обработчик прерывания, давайте теперь научимся его устанавливать.

Установка обработчика прерывания

Регистрация (или установка) подпрограммы обработки прерываний заключается том, что мы помещаем ее начальный адрес в соответствующее поле таблицы векторов прерываний. Это можно сделать двумя способами:

Мы можем просто изменить адрес прерывания таким образом, чтобы он Указывал на наш собственный обработчик и так все и оставить. Однако при этом, старый обработчик прерывания никогда больше вызываться не будет.

Кроме того, а что произойдет, если прерывание будет сгенерировано именно в тот момент, когда мы будем заниматься заменой адреса в таблице векторов? Р-р-раз и «зависли» — вот что случится! Именно поэтому пусть это действие за вас выполнит DOS при помощи функции _dos_setvect(). Для сохранения старого вектора прерывания следует использовать функцию _dos_getvect(). Обе функции гарантировано изменяют вектора без ущерба для операционной системы;

Очень часто нам требуется расширить функциональные возможности системы без полной замены уже работающих функций. Тогда мы используем цепочку прерываний. Под цепочкой прерывания понимается следующее: мы запоминаем старый адрес процедуры обслуживания прерывания и заменяем его на собственный, затем, по окончании нашей процедуры обработки прерывания, передаем управление старому обработчику. Таким образом, нам удается сохранить чужие обработчики прерываний или резидентные программы - если, конечно, нам это надо!

Взгляните на рисунок 12.3, на котором представлены два способа установки обработчиков прерываний.

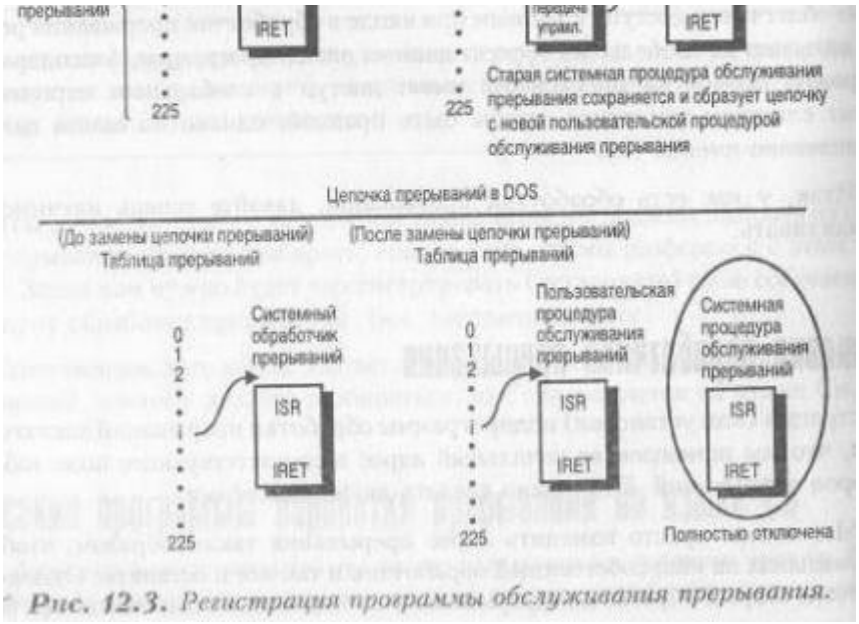


Рис. 12.3. Регистрация программы обслуживания прерывания.

Листинг 12.1 демонстрирует установку нашей процедуры обслуживания прерывания timer () (которая по-прежнему ничего не делает).

Листинг 12.1. Установка процедуры обслуживания прерывания.

```
void (_interrupt _far old_isr) (); // Указатель для сохранения
// вектора старого
// обработчика прерываний
// Сохранить вектор старого обработчика прерывания таймера
old_isr =  _dos_getvect( 0x1C);
// Установить новую процедуру обработки прерывания
_dos_setvect(0x1C, Timer);
```

Ну как, трудно? Да вовсе нет! Для восстановления старого обработчика нам потребуется сделать только:

```
_dos_setvect(0x1C, old_isr);
```

Вот и все. Теперь, для примера, заставим наш обработчик прерывания делать что-нибудь полезное, например, обновлять значения глобальной переменной. В Листинге 12.1 показан текст программы SPY.C, которая представляет собой бесконечный цикл, печатающий значение переменной. Единственный способ изменить значение этой переменной - присвоить ей значение из другой программы, например, из обработчика прерывания.

Листинг 12.2. Шпионим за часами (SPY.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ////////////////////////////////////////////////
#include <dos.h>
#include <bios.h>
#nclude <stdio.h>
#include <math.h>
#include <conio.h>
#include <graph.h>
// ОПРЕДЕЛЕНИЯ//////////////////////////////////
```

```
#define TIME_KEEPER_INT 0x1C
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ ////////////////////////////
void (_interrupt _far *old_Isr)();
// хранит старый обработчик прерывания
long time=0;
// ФУНКЦИИ //////////////////////////////////
void _interrupt _far Timer()
{
// Увеличивает глобальную переменную.
// Еще раз отметим, что мы можем это делать, так как при входе
// в процедуру обработки прерывания регистр DS указывает на сегмент
// глобальных данных нашей программы,
time++;
} // конец Timer
// ОСНОВНАЯ ПРОГРАММА ////////////////////////////////
void main(void)
{
// установка процедуры обработки прерывания
Old_Isr = _dos_getvect(TIME_KEEPER_INT);
_dos_setvect(TIME_KEEPER_INT, Timer) ;
// ожидание нажатия клавиши пользователем
while(!kbhit())
{
// вывод переменной. Примечание: сама по себе функция main
// значение этой переменной не изменяет...
_settextposition(0,0);
printf("\nThe timer reads:%ld   ",time);
} // конец while
// восстановление старого обработчика прерывания
_dos_setvect(TIME_KEEPER_INT, Old_Isr) ;
} // конец функции main
```

Запустив программу, приведенную в Листинге 12.2, вы увидите, что приращение счетчика происходит очень быстро. Фактически, значение этой переменной увеличивается каждую 1/18.2 секунды — так настроены по умолчанию внутренние системные часы. (Не переживайте, очень скоро мы научимся управлять и ими). Главное, что сама программа не увеличивает значенй переменной time. Этим занимается исключительно подпрограмма обслужива ния прерывания.

Итак, мы создали процедуру обслуживания прерывания, установили ее и убедились в том, что она работает - и все это мы сделали, написав лишь несколько строк кода. Восхитительно, не правда ли? Более сложными обработчиками прерываний мы займемся чуть позже. Сейчас же давайте сменим тему и поговорим об игровом цикле.

Игровой цикл

Обучаясь мастерству создания компьютерных игр, вы уже достаточно узнали обо всех необходимых компонентах. Однако у вас, наверное, уже возник вопрос о программной архитектуре игры. В этом разделе я создам каркас типичной компьютерной игры.

Компьютерная игра не так уж и сложна, как это может показаться. Для того чтобы она работала, вы должны:

- Проинициализировать все системы;
- Начать игровой цикл;
- Получить команды игрока;
- Отработать логику игры и проделать все необходимые преобразования участвующих в игре объектов;
- Визуализировать графическое изображение;
- Вернуться к началу цикла.

Вкратце, это все, что вам следует сделать. Впрочем, вы прекрасно представляете, что в деталях каждая из этих задач достаточно сложна. Кроме того, иногда надо сделать так, чтобы некоторые события в игре происходили независимо от логики самой программы. Например, вам может понадобиться изменять значение некоторой переменной строго через 1/30 долю секунды или до перехода к следующему циклу может возникнуть необходимость ожидания синхронизации с некоторым событием (вроде начала вертикального обратного хода луча). Подобные факторы обычно несколько нарушают четкую структуру теоретического цикла игры.

Создавая компьютерную игру, прежде всего следует решить, что это будет за игра, какие существа в ней будут жить, какие в ней будут графика и звук и так далее. Затем следует набросать примерную схему работы игры и примерную схему воплощения игры в строки программы. На этом этапе не надо задумыся о низкоуровневых деталях реализации (таких как способ вывода изображения или принцип работы с драйверами звуковых карт). Нас, прежде всего, интересует общая картина программы.

Никто за вас не решит, как вы будете реализовывать свою игру. Помните что не существует

единственно правильного способа сделать игровую программу. Если, в конце концов, программа будет работать так, как надо — неважно, как именно она реализована. Однако следует иметь в виду, что некоторые способы построения подобных программ лучше других и это «лучше» определяется следующими факторами:

- Время, затраченное на реализацию игры;
- Скорость выполнения программы;
- Простота модификации;
- Легкость отладки.

Этот список можно продолжить, однако это, пожалуй» наиболее важные факторы.

В четырнадцатой главе, «Связь», мы займемся игрой, которую я назвал Net-Tank. Это работающий через последовательный порт симулятор танкового боя, со звуковыми эффектами, потрясающей графикой, различным оружием и полностью интерактивным режимом игры. Цикл этой игры приведен на рисунке 12.4.

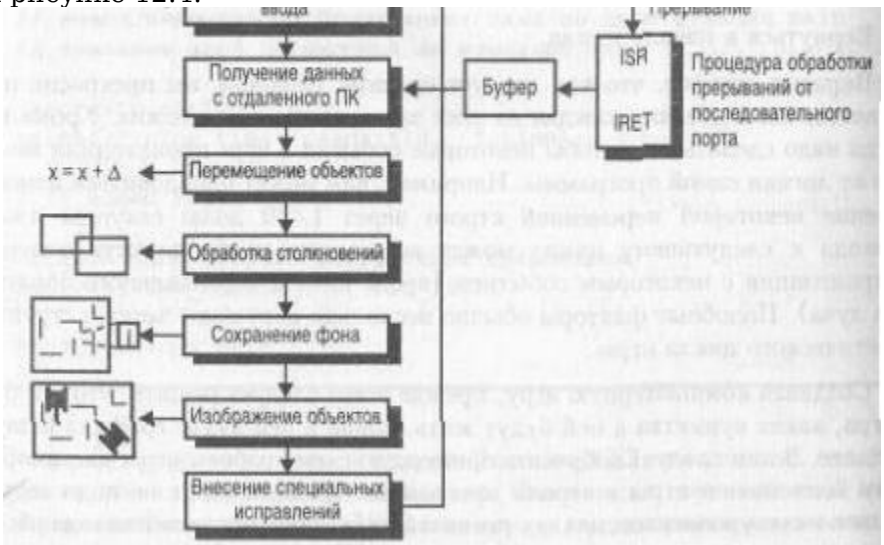


Рис. 12.4. *Пример игрового цикла.*

Как вы можете видеть, цикл игры включает в себя все обсуждавшиеся ранее компоненты:

- Инициализацию объектов игры;
- Удаление объектов;
- Ввод данных локальным игроком;
- Ввод данных удаленным игроком;
- Выполнение преобразований;
- Расчет столкновений;
- Вызов подпрограммы, перемещающей снаряды;
- Воспроизведение графического изображения в автономном буфере экрана. В нужный момент это изображение мгновенно возникнет на экране.

Это хороший пример цикла игры. Он включает в себя все компоненты, присущие более сложным играм. Однако сама игра проста в исполнении и я вполне мог бы обойтись без использования прерываний и синхронизации. (Хотя на самом деле это не так - передачу данных через последовательный порт в любом случае пришлось бы осуществлять при помощи обработчика прерывания, но об этом вы узнаете в четырнадцатой главе). Как бы то ни было, у меня не происходило никаких асинхронных событий, я не создавал сложную систему . воспроизведения звука и был избавлен от необходимости выполнять критичную по времени обработку данных, которая наверняка потребовала бы строгого слежения за временем исполнения посредством использования прерываний.

Теперь давайте поговорим о методах создания программного обеспечения, помогающих избежать некоторых характерных для компьютерных игр проблем. Конечно, можно обойтись и без этих хитростей и уловок, но с ними будет проще быстро написать хорошую игру.

Автономные функции

В компьютерной игре каждую секунду происходят сотни (если не тысячи) самых разнообразных вещей и программа должна следить за множеством различных переменных и состояний. Однако большинство действий, которые происходят во время игры, не нуждаются в жестком контроле со стороны основной программы, поэтому было бы неплохо сделать некоторые функции «самодостаочными» с точки зрения инициализации, синхронизации и тому подобных действий. Я назвал такие функции *автономными.* Я придумал для них такое название с тем, чтобы подчеркнуть присущую им способность совершать операции, не требуя внимания со стороны

программы. Программа в течение цикла игры должна только вызывать эти функции в нужный момент, а все остальное они выполнят сами.

Допустим, например, что мы хотим сделать функцию, которая медленно изменяет содержимое регистра определенного цвета. Этот регистр может использоваться для изображения различных мелких деталей на экране. При выполнении функции изменяется цвет регистра, и таким образом изменяется один из цветов экрана. Результат работы этой функции отражается только на изображении. Функция, решающая подобную задачу идеально приспособлена для автономной работы. В Листинге 12.3 приведен текст программы, реализующей такую функцию.

Листинг 12.3. Автономное управление светом.

```
void Strobe_Lights (void)
{
static clock=0,     // Функция имеет собственные часы, поэтому она
// может инициализироваться при первом вызове
first_time=1
//Проверка на первый вызов
if (first_time)
{
first_time=0; // Сброс флага первого вызова
// Инициализация
} // конец оператора if
else // не первый вызов
{
// пора ли выполнять действия?
if (++clock==100)
change color register
//сброс времени
clock=0
} // конец if
} // конец else
} // конец Strobe_Lights
```

Примечательно то, что функция Store_Lights() самодостаточна. Вызывающая ее функция не должна передавать ей никаких конкретных параметров для того, чтобы она могла инициализироваться и нормально работать. Более того, у этой функции есть свои локальные статические переменные, которые сохраняют свои значения даже после того как функция отработает. Вот в чем суть автономных функций. Программе требуется только вызывать эту функцию в игровом цикле и она может быть уверена, что цвета будут изменяться. При этом программа избавлена от необходимости отслеживать значения каких-либо переменных, инициализировать функцию и так далее.

Все это, конечно, здорово. Однако есть и одна проблема: эта функция аппаратнозависимая. Как вы уже заметили, функция считает до 100 и затей делает то, что надо, но при этом скорость ее счета зависит от быстродействия компьютера. Чем мощнее компьютер, тем быстрее выполняются вычисления. В результате на одних компьютерах цвета будут изменяться быстрее, чем на других. Чтобы избавиться от такого рода проблем, мы должны уметь каким-то образом узнавать время.

Информация о текущем времени очень важна и она может пригодиться нам не только для того, чтобы что-то делать в определенное время, но и для других самых разнообразных целей. Так, например, нам может понадобиться птичка, пролетающая по экрану каждые пять секунд, или обновление изображения не реже тридцати раз в секунду. Чтобы добиться этого, необходимо иметь под рукой какой-то таймер и способ доступа к нему.

Итак, поговорим о функциях ответа, которые можно использовать как для решения данной задачи, так и для многих других целей.

Функции ответа

Название этой функции - *функция* ответа — целиком отвечает ее назначению. Эта функция реагирует на то, что произошло какое-то событие: наступило определенное время, изменилось значение переменной или что-то еще. С помощью функций ответа мы будем реагировать на события или состояния определенными действиями.

Например, с помощью такой функции мы могли бы следить за временем и каждые пять минут воспроизводить какой-либо звук. В данном случае функция ответа реагирует на то, что пять минут прошли, издавая звуковой сигнал. С другой стороны, мы могли бы написать и такую функцию ответа, которая следила бы за переменной, значение коей может меняться либо самой программой либо во время обработки прерывания, и реагировала бы на такое изменение.

На рисунке 12.5 показано взаимодействие функций ответа с другими частями системы.

Все, о чем мы сейчас говорим, относится к методам программирования задач "реального времени". Обычно их не обсуждают ни в книгах, ни в институтах (возможно, поэтому они и кажутся вам несколько странными).

Но не будем отвлекаться и напишем функцию ответа, следящую за переменной timer. Через каждые пять «тиков» системных часов эта функция обнуляет значение переменной timer и изображает в случайной точке экрана пиксель. Текст этой программы приведен в Листинге 12.4.

Рис. 12.5. Функции ответа.

Листинг 12.4. Функция ответа (PRES.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ////////////////////////////////////////
#include <dos.h>
#include <bios.h>
#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <graph.h>
// ОПРЕДЕЛЕНИЯ /////////////////////////////////
#define TIME_KEEPER_INT 0x1C
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ ///////////////////////////////////////////
void (_interrupt _far *01d_Isr)();
// хранит Старый обработчик прерывания
long time=0;
// ФУНКЦИИ ////////////////////////////////////////////
void _interrupt _far*Timer ()
{
// увеличивает глобальную переменную
// еще раз отметим, что мы можем это делать, так как при входе в
// процедуру обработки прерывания регистр DS указывает-на сегмент
// глобальных данных нашей программы
time++;
} // конец Timer
//////////////////////////////////////////
void Plot_Responder(void)
{
static int first_time=l;
static long old_time;
// проверка первого вызова
if (first_time)
{
//сброс флага первого вызова
first_time=0;
old_time = time;
} // конец оператора if else
( // не первый вызов
// прошло ли пять "тиков"?
if ( (time-old_time)>=5)
{
old_time = time; // сохранить новую точку отсчета
// вывод пикселя на экран
_setcolor(rand()%16);
```

```
_setpixel(rand()%320,rand()%200) ;
} // конец оператора if
} // конец else
}// конец Plot_Responder,
// ОСНОВНАЯ ПРОГРАММА ////////////////////////////////////////
main()
{
_setvideomode(_MRES256COLOR);
printf('Hit any key to exit...");
// установка процедуры обработки прерывания
0ld_Isr = _dos_getvect(TIME_KEEPER_INT);
_dos_setvect(TIME_KEEPER_INT, Timer);
// ожидание нажатия клавиши пользователем
while(!kbhit())
// ... текст программы игры
// вызов всех функций ответа Plot_Responder() ;
// ... текст программы игры
}  // конец while
_setvideomode(_DEFAULTMODE) ;
// восстановление старого обработчика прерывания
_dos_setvect(TIME_KEEPER_INT, 01d_Isr) ;
} // конец функции main
```

В программе, приведенной в Листинге 12.4, функция ответа следит за временем и каждые пять «тиков» системных часов или 5x55.4 миллисекунды рисует на экране пиксель. Значение 55.4 получается из того, что системный таймер по умолчанию настроен так, что генерирует прерывание 18.2 раза в секунду или каждые 55.4 миллисекунды. Обратите внимание и на то, что функция ответа автономна.

Автономные функции и функции ответа могут быть использованы для самых разнообразных целей. Они позволяют выделить часть задач, не связанных с непосредственным ходом игры в отдельную группу и решать их отдельно. Логика игры и так достаточно сложна - поэтому не стоит ее без необходимости загромождать мерцающими огоньками и прочими мелкими деталями.

На персональных компьютерах следить за временем и,управлять событиями связанными с ним, можно только с помощью внутренних таймеров системы. К сожалению, большинство из них используется для других целей. Однако один применяется специально для внутренних часов компьютера. Он «тикает» со тростью 18.2 раза в секунду. Что и говорить, необычная отправная точка для вычислений и, честно говоря, я понятия не имею, почему создатели персональных компьютеров выбрали именно это значение. Но мы можем перепрограммировать его и задать более разумное число, например, 20, 30 или 60 «тиков» в секунду. Естественно, такие значения удобнее использовать в нашей компьютерной игре. Итак, давайте разберемся, как можно перепрограммировать внутренний таймер системы.

Программируем системные часы

Внутренние часы компьютера, следящие за временем, на самом деле вовсе никакие не часы. Это микросхема типа таймер/счетчик, которая просто в, зависимости от начальных настроек с определенной частотой изменяет значение счетчика. Эта микросхема называется микросхемой таймера 8253. В ней содержится три шестнадцатиразрядных счетчика. Эти счетчики можно программировать самыми разнообразными способами. В таблице 12.2 показано назначение каждого из них.

Таблица 12.2. Счетчики микросхемы таймера 8253.

Порт ввода/вывода	Номер счетчика	Назначение
40h	Счетчик 0	Таймер/Диск
41h	Счетчик 1	Обновление памяти
42h	Счетчик 2	Накопитель на магнитной ленте
43h	Управляющий регистр	Управление таймером

Доступ к микросхеме таймера осуществляется через порты 40h-43h. Как видите, мы можем использовать только счетчики 0 и 3. Связываться со счетчиком 1 нам определенно не стоит! Счетчик 0 уже используется DOS для системных часов. Так почему бы нам не использовать его, перепрограммировав на нужную частоту?

Это вполне можно сделать. И этот метод не вызовет неполадок в компьютере. Единственный недостаток такого подхода состоит в том, что изменится системное время. Мы можем обойти это, сохранив время, за которое изменяется значение частоты таймера и по ходу игры отслеживать время с учетом этого изменения. Затем, перед окончанием работы программы, следует

восстановить прежнее значение частоты и изменить системное время. Для сохранения и восстановления системного времени можно использовать функции Си.

Изменение частоты таймера

Давайте поговорим о том, как изменять частоту таймера. Таймер генерирует прерывания следующим образом: заданное значение регистра счетчика таймера уменьшается на единицу каждый раз, когда приходит импульс от генератора. Когда значение счетчика становится равным нулю, таймер генерирует прерывание, после чего восстанавливает исходное значение счетчика. Затем процесс повторяется вновь. Счетчик 0 генерирует прерывания 18.2 раза в секунду или через каждые 55.4 миллисекунды. Это время задается следующим образом: на микросхему таймера подаются импульсы с частотой 1.19318МГц. В случае системного таймера в регистр помещается число 65536. Если взять 1.19318 и разделить па 65536 получится 18.206 циклов в секунду или 54.925 миллисекунд. Как видите, создатели персонального компьютера просто использовали максимально возможный делитель, который может поместиться в 16 битов: 65535. (На самом деле таймер «тикает» на один раз больше — вот откуда я взял число 65536).

Таким образом, все что нам следует сделать, это вычислить 16-разрядное число, при делении на которое 1.19318МГц даст нам нужную частоту. Для вашего удобства я рассчитал подходящие значения для наиболее часто встречающихся частот. Взгляните на таблицу 12.3

Таблица 12.3. 16-разрядные значения счетчика для наиболее употребительных частот таймера.

Шестандцатиричное Значение	**Десятичное Значение**	**Частота**
4DAEh	19886	60Hz
965C	39722	30Hz
E90B	59659	20Hz

Таким образом, нам следует перепрограммировать счетчик 0 таким образом, чтобы он генерировал прерывания с более подходящей для нашей игры частотой - так, чтобы наша игра могла контролировать события чаще чем 18.2 раза в секунду.

Давайте посмотрим, каким образом можно задать соответствующее значение счетчика. Нас интересуют порты ввода/вывода 40h и 43h (все, о чем мы будем говорить, относится к ним): счетчик 0 и соответствующий управляющий регистр. Значения битов управляющего регистра, показаны в таблице 12.4.

Таблица 12.4. Назначение битов управляющего регистра микросхемы 8253.

Бит 0 = 0: Значение счетчика в двоичном коде.

Бит 0 = 1: Значение счетчика в двоично-десятичном коде.

Биты 1–3 используются для выбора режима работы счетчика.

Бит 3	Бит 2	Бит 1	Режим
0	0	0	0 — Прерывание по счетчику терминала
0	0	1	1 — Аппаратно переключаемый одиночный импульс
X	1	0	2 — Генератор частоты
X	1	1	3 — Генератор *прямоугольных* импульсов
1	0	0	4 — Программно формируемый строб
1	0	1	5 — Аппаратно формируемый строб

Биты 4 и 5 управляют чтением и записью.

Бит 5	Бит 4	Функция
0	0	Операция блокировки *счетчика* (передать значение счетчика в задвижку)
0	1	Прочитать/записать младший байт счетчика
1	0	*Прочитать/записать* старший байт счетчика
1	1	Прочитать/записать младший байт, а затем старший байт счетчика

Биты 6 и 7 выбирают счетчик для выполнения операции.

Бит 6	Бит 7	Номер счетчика
0	0	0
0	1	1
1	0	2
1	1	Запрещено

Из таблицы 12.4 видно, что в нашем распоряжении много параметров. Впрочем, нам сейчас все они не нужны, нам надо всего-навсего изменить начальное значение счетчика 0. При этом значения других битов должны быть следующими:

- Режим работы - 2, генератор частоты;
- Метод подсчета - двоичный;
- Запись регистра счетчика осуществляется при помощи операции «Прочитать/записать младший байт, а затем старший байт счетчика».

Таким образом, все что мы должны сделать, это записать управляющее слово в порт 43h, а затем выполнить две операции записи в порт 40h. В ходе первой операции записи мы установим значение младшего байта нового значения счетчика, а при второй — значение старшего байта. В том, где какие байты аппаратные средства разберутся сами, поэтому вам об этом волноваться не надо. В Листинге 12.5 приведен текст функции, программирующей таймер.

Листинг 12.5. Перепрограммируем системные часы (OUTATIME.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ////////////////////////////////////
#include<stdio.h>
#include<conio.h>
// ОПРЕДЕЛЕНИЯ /////////////////////////////////////////
#define CONTROL_8253 0x43 // управляющий регистр 6253
#define CONTROL_WORD 0x3C // управляющее слово, задающее режим 2,
```

```
// двоичный подсчет, запись
// младший/старший байт
#define COUNTERED     0x40 // счетчик 0
#define TIMER_60HZ    0x4DAE // 60Гц
#define TIMER_30HZ    0x965C // 30Гц
#define TIMER_20HZ    0xE90B // 20Гц
#define TIMER_18HZ    0xFFFF // 18.2Гц (стандартная частота)
// МАКРОСЫ ////////////////////////////////////////////
#define LOW_BYTE(n) (n & 0x00ff)
#define HI_BYTE(n) ((n>>8) & 0x00ff}
// ФУНКЦИИ ////////////////////////////////////////////
void Change Time(unsigned int new count)
{
// послать управляющее слово, задающее режим 2, двоичный подсчет,
// запись младший/старший'байт

_outp(CONTROL_8253, CONTROL_WORD);
// теперь запишем младший значащий байт в регистр счетчика
_outp(COUNTER_0,LOW_BYTE(new_count));
//я теперь запишем старший байт в регистр счетчика
_outp(COUNTER_0,HI_BYTE(new_count) ) ;
} // конец Change_Time
// ОСНОВНАЯ ПРОГРАММА ////////////////////////////////
main()
{
// перепрограммирование таймера с частоты 18.2Гц на частоту 60Гц
Change_Time (TIMER_60HZ);
} // конец,функции main
```

После окончания этой программы системный таймер будет работать слишком быстро для внутренних часов DOS. Для того чтобы это поправить, вы можете:

- Перезагрузить компьютер;
- Запустить программу еще раз, изменив константу, передаваемую функции Change_Time (), на TIMER_18HZ.

Собираем все вместе

Итак, мы рассмотрели основы работы с прерываниями, многозадачность и некоторые приемы программирования, которые несколько облегчают реализацию архитектуры компьютерной игры. Причем представленная здесь коллекция приемов программирования далеко не самая полная и лучшая. Мы просто обсудили эти вопросы для того, чтобы вы смогли представить себе различные способы организации игры. Впрочем, вскоре вы, наверняка, придумаете и свои Методы и трюки. Главная цель этой главы состоит в том, чтобы дать вам некую Справную точку и направление для ваших размышлений. Теперь я хочу просуммировать все полученные вами знания и разобрать несколько примеров применения рассмотренных нами методов. После этого для окончательного закрепления материала я приведу еще два примера программ.

В этой главе мы узнали о реализации многозадачности на персональных компьютерах с помощью прерываний. Прерывание — это всего-навсего мгновенная передача управления от выполняющейся программы к процедуре обслуживания прерывания (ISR), которая выполняет все действия, относящиеся к событию, вызвавшему прерывание. Прерывания полезны тем, что они позволяют реализовать на персональном компьютере некое подобие многозадачности. Более того, они могут быть использованы для управления задачами, зависящими от времени и событий, контролировать которые главной программе крайне затруднительно именно в силу самой природы этих задач. Обычно процедуры обслуживания прерываний используются для таких вещей как обработка ввода с клавиатуры или передача данных через последовательный порт. Однако мы можем использовать их и для других задач (работа со звуком, система ввода/вывода, обслуживающие функции и так далее).

Управление задачами, зависящими от времени и событий, осуществляется за счет подпрограмм обработки тех прерываний, которые происходят по наступлению определенного момента времени или в результате какого-либо события. В этом смысле очень полезным оказывается прерывание системного таймера 0x1C. Мы можем поместить адрес нашего обработчика в таблицу векторов по адресу, соответствующему прерыванию, которое генерируется при каждом системном «тике» (то есть 18.2 раза в секунду). Это гарантирует нам, что независимо от загруженности системы, наша подпрограмма обработки прерываний будет вызываться всегда с неизменной частотой. Кроме того, мы знаем, что системный таймер можно запрограммировать и на другую частоту, например, 20Гц, 30Гц, 60Гц и так далее. Нам это может понадобиться, если мы собираемся использовать прерывание для чего-то, что требует более частого выполнения.

И, наконец, мы узнали, как использовать прерывания для изменения глобальных переменных, являющихся, по сути, сообщениями, на которые могут реагировать другие функции той же программы. Это может пригодиться в тех случаях, когда в игре надо выполнять критичные по времени исполнения операции, синхронизацию событий и тому подобные задачи.

Кроме того, мы обсудили как построить цикл игры. На самом деле это просто один из методов организации программы и входящих в нее функций таким образом, что все действия выполняются в определенной последовательности и в соответствии с определенной логикой. Мы узнали, что программа игры должна иметь раздел инициализации, за которым следует главный цикл событий. Внутри этого цикла осуществляется стирание графического изображения, получение входных данных от пользователя, выполнение необходимых графических преобразований, формирование и вывод на экран графического изображения. Конечно, порядок действий может быть несколько иным, как, впрочем, и входящие в него элементы, однако нам всегда следует создавать цикл событии, внутри которого эффективно и логически обоснованно делается все, что должно быть сделано.

Рассмотрев цикл игры, мы перешли к обсуждению достаточно отвлеченных на первый взгляд вопросов, относящихся к приемам программирования. Мы изучили два тесно связанных между собой типа функций, которые назвали автономными функциями, и функциями ответа. Эти приемы программирования были приведены в качестве иллюстрации, как много действий и процессов в компьютерной игре могут быть организованы независимо друг от друга и иметь свои собственные данные. Компьютерные игры настолько сложны, что нам часто приходится писать функции, работающие без постоянного контроля со стороны главной программы. В этом нам помогают автономные функции. Кроме того мы рассмотрели еще и функции ответа. Эти функции «откликаются» на определенные события соответствующими действиями.

Обсуждение всех этих вопросов призвано помочь вам осознать, как писать программы, работающие в «реальном режиме времени». Иными словами, как организовать выполнение нужных действий таким образом, чтобы создать впечатление одновременности происходящего. В компьютерных играх без этого не обойтись.

Напоследок мы узнали, как перепрограммировать системные часы, реализованные микросхемой таймера 8253. Мы узнали, что в счетчике 0 хранится 16-разрядное значение, которое рассматривается как делитель поступающего сигнала с частотой 1.19318МГц. Результирующий сигнал используется для генерации «тиков», своего рода пульса, который инициирует прерывание по времени. Это прерывание может быть перехвачено с помощью вектора прерывания 0x1C. В ста процентах игр этот таймер перепрограммируется на более подходящую частоту вроде 30Гц или 60Гц. Таким образом, работа программы синхронизируется по новой базовой частоте. Помимо этого, входящая в состав игры подсистема вывода звуковых эффектов и подсистема ввода/вывода реа лизуются в виде фоновых задач, которые реализованы в виде процедур обслуживания таймерного прерывания 0x1C. Благодаря этому ввод/вывод и музыка исполняются с неизменной скоростью даже тогда, когда сама игра несколько замедляется вследствие выполнения вычислений или сложных графических эффектов.

Писать эту главу было непросто, поскольку в ней мне пришлось изложить вам очень много новых вещей. На самом деле, следует иметь в виду, что только опыт поможет вам разобраться с миллионами тонкостей, которые приходится учитывать при программировании компьютерных игр. Тем не менее, перед вами уже открылись новые возможности. Сейчас вы готовы мыслить по-новому и это Уже неплохо.

Как я и обещал, сейчас мы напишем еще несколько подпрограмм обслуживания прерываний и сделаем с их помощью несколько потрясающих вещей.

Пример обработчика прерывания № 1 - Там полно звезд...

Чтобы продемонстрировать вам, насколько полезными могут оказаться многозадачность и прерывания, я написал в виде обработчика прерывания программу, которая рисует трехмерное звездное небо. Как вам уже известно, при каждом приращении счетчика внутреннего таймера, генерируется прерывание. При помощи вектора 0x1C я сопоставил этому прерыванию свою процедуру обслуживания, которая и создает трехмерное звездное небо. Поскольку обращение к моему обработчику происходит 18.2 раза в секунду, значит и картина звездного неба обновляется с этой же частотой. Нечто подобное мы могли бы использовать при написании игры, работающей на одном экране (игры типа «перестреляй их всех»). Наше звездное небо будет изменяться независимо от того, что будет происходить на переднем плане.

Обратите внимание на то, что процедура обслуживания прерывания сделана автономной. Это означает, что она инициализирует свои данные при первом обращении к ней (в этот момент она создает базу данных с информацией о звездах), а при последующих вызовах просто видоизменяет картинку.

И напоследок еще одно маленькое замечание: завершая программу, вы можете оставить ее резидентной в памяти, нажав клавишу E. Сделав это, вы увидите, что звездное небо присутствует в строке приглашения DOS, что выглядит несколько странно! Причина такого поведения программы кроется в том, что когда вы завершаете ее нажатием клавиши E, она не восстанавливает прежний обработчик прерывания. Компьютер продолжает вызывать по прерыванию от таймера наш обработчик. Вы наверняка даже и представить себе не могли, что это сработает! И, тем не менее, завершаясь, программа оставляет в оперативной памяти большую часть своего кода неизменным, в результате чего обработчику прерывания удается пережить окончание работы породившей его программы. Если вы попытаетесь запустить еще какую-нибудь программу, компьютер, скорее всего, «зависнет». Поэтому для написания настоящих резидентных программ такой метод применять не стоит. (На самом деле для этих целей предназначена специальная функция DOS, которая называется dos keep, но сейчас мы не будем подробно рассматривать резидентные программы. Мы просто случайно создали одну из них). Если система «зависнет», вам придется перезагрузиться.

Текст программы, изображающей трехмерное звездное небо, представлен в Листинге 12.6.

Листинг 12.6. Трехмерное звездное небо (STARS.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ////////////////////////////////////
#include <dos.h>
#include <bios.h>
#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <graph.h>
// ОПРЕДЕЛЕНИЯ /////////////////////////////////////////
#define TIME_KEEPER_INT 0x1C
#define NUM_STARS 50
// СТРУКТУРЫ ///////////////////////////////////////////
typedef struct star_typ
int x,y;    // координаты звезды
int vel;    // проекция скорости звезды на ось X
int color; // цвет звезды
} star, *star_ptr;
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ ///////////////////////////////
void (_interrupt _far *01d_Isr)(void);
// хранит старый обработчик прерывания
unsigned char far *video_buffer = (char far *)0xA0000000L;
// указатель на видеобуфер
int star_first=1; // флаг первого вызова автономной функции
// звездного неба
star starstNUM_STARS]; //звездное небо
// ФУНКЦИИ /////////////////////////////////////////////////
void Plot_Pixel_Fast(int x,int y,unsigned char color)
{
// эта функция рисует точку заданного цвета несколько быстрее,
// чем обычно за счет применения операции сдвига вместо операции
// умножения
// используем известный факт, что 320*y = 256*y + 64*y = y<<8 + y<<6
video_buffer[((y<<8) + (y<<6)) + x] = color;
} // конец Plot Pixel_Fast
//////////////////////////////////////////////////////////
void _interrupt _far Star_Int(void)
{
// эта функция создает иллюзию трехмерного звездного неба,
// наблюдаемого из иллюминатора космического корабля Enterprise
// Замечание: эту функцию следует исполнять не реже одного раза
// в 55.4мс, иначе не избежать перезагрузки!
int index;
// проверка, надо ли нам инициализировать звездное поле
// (то есть первый ли раз вызывается функция)
if (star_first)
{
// сброс флага первого вызова
star_first=0;
// инициализация всех звезд
for(index=0; index<NUM_STARS; index++)
{
// инициализация цвета, скорости и координаты каждой звезды
stars[index].x     = rand()%320;
stars[index].y     = rand()%180;
// выбор плоскости для звезды switch(rand()%3)
{ case 0:
// плоскость 1 - наиболее удаленная плоскость
{
// установка скорости и цвета звезды
stars[index].vel = 2;
stars[index].color = 8;
} break;
case 1: // плоскость 2 - плоскость, расположенная
// посередине
{
stars[index].vel = 4;
stars [index] .color =7;
```

```
} break;
case 2: // плоскость 3 -самая ближняя плоскость
{
stars[index].vel = 6;
stars[index].color = 15;
} break;
} // конец оператора switch
} // конец цикла
} // конец оператора if
else
{ // не первый вызов функции, поэтому производим рутинные
// действия: стирание, перемещение, рисование
for (index=0; index<NUM_STARS; index++)
{ // стирание
Plot_Pixel_Fast(stars[index].x,stars[index]-y,0);
// перемещение
if ((stars[index].x+=stars[index].vel) >=320 ) stars[index].x = 0;
// рисование
Plot_Pixel_Fast(stars[index],x,stars[index],y, stars[index].color);
} // конец цикла
} // конец else
} // конец Star_Int
// ОСНОВНАЯ ПРОГРАММА ///////////////////////////////////
void main(void)
{ int num1, num2,c ;
_setvideomode(_MRES256COLOR) ;
// установка обработчика прерывания
Old_Isr = _dos_getvect(TIME_KEEPER_INT) ;
_dos_setvect(TIME_KEEPER_INT, Star_Int);
// ожидание нажатия клавиши пользователем
_settextposition(23,0);
printf("Hit Q - to quit.");
printf("\nHit E - to see something wonderful...");
// чтение символа
c = getch();
// хочет ли пользователь рискнуть?
if (c=='e')
{
printf("\nLook stars in DOS, how can this be ?") ;
exit(0);
// выход без восстановления старого обработчика прерывания
} // конец оператора if
/ восстановление старого обработчика прерывания
_dos_setvect(TIME_KEEPER_INT, 01d_Isr);
_setvideomode(_DEFAULTMODE) ;
}// конец функции main
```

Пример обработчика прерывания № 2 - Ловим нажатия клавиш!

Как мы узнали в третьей главе, «Основы работы с устройствами ввода», мы можем использовать BIOS для чтения нажатия клавиши путем проверки значения скан-кода. Это отличный способ, и он вполне применим для большинства случаев, но что делать, если вы хотите одновременно нажать две клавиши? (Примером такой ситуации может служить момент, когда вы одновременно нажимаете клавиши «стрелка вверх» или «стрелка вниз» и «стрелка вправо» чтобы изменить направление движения на диагональное). Единственный способ обработать подобную ситуацию в программе требует более тонкой работы с клавиатурой. BIOS по сравнению со стандартными функциями Си просто дает нам еще один уровень управления, но если мы хотим добиться необходимой для профессионально написанных компьютерных игр функциональности, нам следует глубже, глубже и еще глубже разобраться с клавиатурой.

На самом деле клавиатура персонального компьютера представляет собой отдельный микрокомпьютер, который называется 8048. При нажатии или отпускании клавиши клавиатура посылает последовательность сигналов персональному компьютеру на обработку. BIOS сообщает нам о том, какая клавиша нажата, но умалчивает об отпущенных клавишах. Такой половинчатой информации не достаточно для того, чтобы организовать обработку одновременного нажатия нескольких клавиш. Нам нужно знать и когда была нажата, и когда была отпущена та или иная клавиша, чтобы написать алгоритм, способный отслеживать состояния некоторого набора клавиш. При нажатии клавиши мы выставляем соответствующий ей флаг в еостояние «нажато», а при отпускании — сбрасываем значение этого флага. Применяя такой подход, мы регистрируем нажатия различных клавиш и храним эту информацию в нашей структуре данных до тех пор, пока

клавиша не будет отпущена.

При нажатии клавиши клавиатурой посылается код нажатия, а при отпускании — соответствующий код отпускания. Эти коды похожи на уже известный вам скан-код. Различие состоит в том, что при отпускании клавиши, каким бы ни было значение кода нажатия, к нему всегда прибавляется 128.

Чтобы наша программа смогла взять контроль за клавиатурой на себя, мы выгрузим обработчик клавиатуры операционной системы DOS и установим наш собственный драйвер. Он будет получать коды клавиш независимо от того, что произошло — нажатие или отпускание, а затем сохранять эту информацию в глобальной переменной, к которой имеет доступ наша Си-программа. При таком подходе функция нашей программы сможет использовать текущее значение этой переменной для выяснения того, какие клавиши в данный момент нажаты, а какие отпущены. Разобравшись что к чему, функция сведет эту информацию в таблицу. Клавиатурное прерывание имеет номер 0x09. Все, что нам требуется сделать, это написать и установить соответствующую процедуру обработки данного прерывания.

Прежде чем мы этим займемся, вспомним адреса клавиатурных портов ввода/вывода и их функции. Собственно клавиатурный порт ввода/вывода находится по адресу 60h, а регистр, управляющий клавиатурой — по адресу 61lh Эти порты находятся на микросхеме PIA (Peripheral Interface Adapter). Кроме всего прочего, нам еще потребуется выполнить определенные процедуры перед вызовом нашего обработчика прерывания и после его завершения. Общий порядок всех действий таков:

1. Войти в процедуру обслуживания прерывания. Это происходит при каяждом нажатии клавиши.
2. Прочитать из порта ввода/вывода 60h код клавиши и поместить его в глобальную переменную для последующей обработки программой или обновления содержимого таблицы, в которой хранится информация о нажатых клавишах.
3. Прочитать содержимое управляющего регистра из порта ввода/вывода 61h и выполнить над ним логическую операцию OR с числом 82h.
4. Записать полученный результат в порт регистра управления 61h.
5. Выполнить над содержимым управляющего регистра логическую операцию AND с числом 7Fh. Это сбрасывает состояние клавиатуры, давая ей понять что нажатие на клавишу обработано и мы готовы к считыванию информации о нажатии других клавиш.
6. Сбросить состояние контроллера прерываний 8259. (Без этого можно и обойтись, однако лучше подстраховаться). Для этого следует записать в порт 20h число 20h. Забавное совпадение, не правда ли?
7. Выйти из обработчика прерывания.

Листинг 12.7 содержит текст программы обработчика клавиатурного прерывания, позволяющего отслеживать состояние клавиш со стрелками. При работе этой программы вы можете использовать курсорные клавиши или их комбинации для перемещения маленькой точки в любом направлении по экрану.

Листинг 12.7. Киберточка (CYBER.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ/////////////////////////////////////
#include <dos.h>
#include <bios.h>
#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <graph.h>
// ОПРЕДЕЛЕНИЯ //////////////////////////////////////////
#define KEYBOARD_INT    0x09
#define KEY_BUFFER      0x60
#define KEY_CONTROL     0x61
#define INT_CONTROL     0x20
// коды нажатия и отпускания для клавиш со стрелками
#define MAKE_RIGHT      77
#define MAKE_LEFT       75
#define MAKE_UP         72
#define MAKE_DOWN       80
#define BREAK__RIGHT     205
#define BREAK_LEFT     203
#define BREAK_UP        200
#define BREAK_DOWN      208
// индексы в таблице состояния клавиш со стрелками
#define INDEX_UP        0
#define INDEX_DOWN      1
#define INDEX_RIGHT     2
#define INDEX_LEFT      3
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ ////////////////////////////////
```

```
void (_interrupt _far *01d_Isr)(void);
// хранит старый обработчик прерывания
unsigned char far *video_buffer = (char far *)0xA0000000L;
// указатель на видеобуфер
int raw_key;    // необработанные данные от клавиатуры
int key_table[4] = {0,0,0,0};
// таблица состояний клавиш со стрелками
// ФУНКЦИИ ////////////////////////////////////////////
void Plot_Pixel_Fast(int x,int y,unsigned char color)
{
// эта функция рисует точку заданного цвета несколько быстрее,
// чем обычно за счет применения операции сдвига вместо операции
// умножения
// используем известный факт, что 320*y = 256*y + 64*y = y<<8 + y<<6
video_buffer[((y<<8) + (y<<6) ) + x] = color;
} // конец Plot_Pixel_Fast
////////////////////////////////////////////
void Fun_Back(void)
{
int index;
// несомненно  запоминающийся рисунок фона
_setcolor(1) ;
_rectangle(_GFILLINTERIOR, 0,0,320,200);
_setcolor{15) ;
for (index=0; index<10; index++)
{
_moveto(16+index*32,0);
_lineto(16+index*32,199);
} // конец цикла
for (index=0; index<10; index++)
{
_moveto(0,10+index*20) ;
_lineto(319,10+index*20);
} // конец цикла
} // конец Fun Back ////////////////////////////////////
void _interrupt _far New_Key_Int(void)
{
// я в настроении немного попрограммировать на ассемблере!
_asm{
sti                ; разрешаем прерывания
in al,KEY BUFFER   ; получаем нажатую клавишу
xor ah,ah          ; обнуляем старшие 8 бит регистра AX
mov raw_key, ax  ; сохраняем код клавиши
in al,KEY_CONTROL ; читаем управляющий регистр
or al,82h       ; устанавливаем необходимые биты для сброса FF
out KEY_CONTROL,al ; посылаем новые данные в управляющий регистр
and al,7fh
out KEY_CONTROL,al ; завершаем сброс
mov al,20h
out INT CONTROL,al; завершаем прерывание
} // конец ассемблерной вставки
// теперь вернемся К Си, чтобы изменить данные
// в таблице состояния клавиш со стрелками
// обработка нажатой клавиши и изменение таблицы
switch(raw_key)
{
case MAKE_UP:    // нажатие стрелки вверх
{
key_table[INDEX_UP]    = 1;
} break;
case MAKE_DOWN:  // нажатие стрелки вниз
{
key_table[INDEX_DOWN]  = 1;
) break;
case MAKE_RIGHT: // нажатие' стрелки вправо
{
key_table[INDEX_RIGHT] = 1;
```

```
} break;
case MAKE_LEFT:  // нажатие стрелки влево
{
key__table[INDEX_LEFT]  = 1;
} break;
case BREAK_UP:    // отпускание стрелки вверх
{
key_table[INDEX_UP]    = 0;
} break;
case BREAK DOWN:  // отпускание стрелки вниз
{
key_table[INDEX_DOWN]  = 0;
} break;
case BREAK_RIGHT; // отпускание стрелки вправо
{
key_table[INDEX_RIGHT] = 0;
} break;
case BREAK_LEFT:  // отпускание стрелки влево
{
key_table[INDEX_LEFT]  = 0;
} break;
default: break;
} // конец оператора switch
} // конец New_Key_Int
//  ОСНОВНАЯ ПРОГРАММА ////////////////////////////////
void main(void)
{
int done=0, x=160, y=100;// флаг выхода и координаты точки
//установка видеорежима 320x200x256
_setvideomode(_MRES256COLOR) ;
Fun_Back(); // оригинальная картинка, не так ли?
printf("\nPress ESC to Exit.");
// установка нового обработчика прерывания
Old_Isr = _dos_getvect(KEYBOARD_INT) ;
_dos_setvect(KEYBOARD_INT, New_Key_Int);
// основной цикл
while(!done)
{
_settextposition(24,2) ;
printf("raw key=%d   ",raw_key);
// смотрим в таблицу и перемещаем маленькую точку
if (key_table[INDEX_RIGHT]) x++;
if (key_table[INDEX_LEFT]) x--;
if (key_table[INDEX_UP]) y--;
if (key_table[INDEX_DOWN]) y++;
// рисуем киберточку
Plot_Pixel_Fast(x,y,10);
// Наша клавиша завершения. Значение кода нажатия ESC равно 1
if (raw_key==1) done=l;
} // конец while
// восстановление старого обработчика прерывания
_dos_setvect(KEYBOARD_INT, Old_Isr) ;
_setvideomode (_DEFAULTMODE) ;
} // конец функции main
```

Уфф... Вот и все, ребята!

Заключение

В этой главе мы изучили некоторые нетривиальные приемы управления работой персонального компьютера. Мы узнали, как работает таймер, как можно использовать прерывания (включая и клавиатурные) и многое другое.

Мы научились писать обработчики прерываний на языке Си и регистрировать их в таблице векторов прерываний. Кроме того, мы узнали много полезного о том, как при помощи автономных функций и функций ответа можно организовать на персональном компьютере синхронизацию программы и обработку наступления определенных событий.

И, в конце концов, мы использовали полученные знания Для написания нескольких

программ, которые брали на себя управление компьютером - одна из этих программ даже не пожелала вернуть управление обратно!

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ

Человеческий разум представляет собой бездну, заполненную прошлым, настоящим и будущим. Он осознает свое собственное существование и является в известном смысле всего лишь устройством с возможностью самоорганизации и способностью задавать вопросы о своем самоощущении. Вычислительная техника находится на пороге понимания механизмов мысли, памяти и сознания. Однако нам, как программистам компьютерных игр, на самом деле не нужно быть слишком искушенными в создании разумно мыслящих систем. Существа в наших играх должны только производить впечатление, что они способны мыслить самостоятельно. В этой части мы избороздим те воды, которые и до сих пор отсутствуют на картах даже самых передовых видеоигр. Мы обсудим темы, в коих область от практического до теоретического полностью эзотерична. Несмотря на это, комплекс мыслящих моделей может быть получен даже с минимальными знаниями. Эти модели будут способны самостоятельно выполнять такие простейшие задачи как сражение, поиск и уклонение.

Мы обсудим следующие темы:

- Обзор того, как мыслят видеоигры;
- Алгоритмы Преследования и Уклонения;
- Шаблонные мысли;
- Случайные передвижения;
- Автоматы с конечными состояниями;
- Вероятностные автоматы;
- Память и обучение;
- Алгоритмы Поиска;
- Теория игр.

Как мыслят видеоигры: обзор

Придет тот день, когда компьютер станет таким же серьезным собеседником, как и лучшие человеческие умы. Однако сегодняшние компьютеры (по крайней мере, ПК) не обладают сложностью, необходимой для инициации мыслей и творческих процессов. Но на самом деле нас это и не должно волновать! Мы же делаем видеоигры, а не андроидов. В игре у нас присутствуют некоторые создания и объекты. Все, что нам нужно сделать, это придать им видимость способности разумного мышления. Играющий может ощущать себя помещенным на короткое время в некое пространство, где он думает, что вражеская атака на корабль реальна! Для осуществления этих целей мы должны проанализировать, каким разумом мы должны наделить наши игровые объекты. Сложность этого «разума» зависит от того, какую разновидность существ мы конструируем. К примеру, создания из Pac Man большую часть своего времени тратят на преследование или убегание от вас. Будь мы теми людьми, что написали Pac Man, мы имели бы возможность загорать где-нибудь на Гавайях, но у нас, в таком случае уже был бы алгоритмический инструмент для осуществления этих преследований и убеганий.

С другой стороны, если бы мы создавали игры, подобные «Космическим Захватчикам», придание видимости интеллекта вряд ли потребовало бы более пары дюжин строк в программе. Псевдокод выглядел бы примерно так:

1. Продолжать движение в том направлении, в котором вы двигаетесь (вправо или влево);
2. Когда вы попадете на границу экрана, изменить направление и двигаться вдоль оси высоты;
3. Перейти к п.1.

Не шибко обширный набор «интеллектуальных» функций для игры, прибыль от которой за все время ее существования составила от 50 до 100 миллионов долларов. Трехмерные видеоигры нам интересны немногому, что они «умнее» обычных «плоских» игр. Объемность дает только повышенное ощущение реальности, а алгоритмы, наделяющие персонажей разумом, в них те же, что и в двухмерных играх. К примеру, Terminator Rampage имеет действительно несложную систему искусственного интеллекта, примерно того же уровня сложности, что и в Pac Man. Однако при наличии трехмерной графики вместе с изощренным звуком существа в ней кажутся вполне одушевленными.

Итак, приведем к общему знаменателю характер большинства компьютерных игр: интеллект персонажей находится на пещерном уровне, но этот недостаток компенсируется за счет графики и звука.

Разработчиков видеоигр идея искусственного интеллекта притягивает снова и снова на протяжении длительного времени, потому что игры, которые они пишут, в большинстве своем довольно агрессивны. Наша задача в этой части

состоит в том, чтобы понять методы, используемые для моделирования интеллекта, и главные категории, внутри которых существуют «мыслящие» алгоритмы. Запомните: игры не думают. Они только «делают умное лицо». Посему большую часть времени мы будем конструировать основной разумный вид деятельности вроде преследования и нападения. Как вы догадываетесь, на это большого ума не надо: обнаружить противника и уничтожить его, или, с точки зрения компьютерной программы, двигаться по направлению к врагу, взять его на мушку и открыть огонь из всех видов оружия.

Алгоритмы Преследования и Уклонения

Итак, начнем. Наиболее простыми игровыми алгоритмами искусственного интеллекта являются так называемый Алгоритм Преследования и его противоположность — Алгоритм Уклонения. В основном, они заставляют игровой объект или догонять игрока, или убегать от него. Конечно, конфигурация игрового пространства также должна учитываться, чтобы преследователь даже и не пытался пройти сквозь стены. Давайте рассмотрим процесс, моделирующий погоню некоторого существа за игроком.

Преследование

Во-первых, нам необходимо знать расположение обоих объектов. У нас есть эти данные, так как мы знаем координаты игрока и игрового объекта, являющегося врагом.

Во-вторых, нам необходимо сконструировать алгоритм, который будет управлять поведением врага, преследующего игрока. Алгоритм 13.1 делает именно то, что мы от него и хотим:

Алгоритм 13.1. Алгоритм Преследования.

```
// Предположим, что px,py - координаты игрока,
// ex,ey - координаты противника
while (игра)
(
программный код
// Вначале рассматриваем перемещение по горизонтали (ось X)
if ex>px then ex=ex+l if ex<px then ex=ex-l
 // Теперь рассматриваем вертикальный (Y) компонент
if ey>py then ey=ey+l if ey<py then ey=ey-l
программный код
}
```

- Если враг находится справа, то есть его X-координата больше соответствующей координаты игрока, преследователь уменьшает свою собственную позицию по оси X на единицу и тем самым приближается к игроку. Если же противник находится левее игрока, он, наоборот, увеличивает свою горизонтальную координату.
- Та же самая логика применима и к перемещению по вертикали.

Используя этот алгоритм, противник преследует игрока почти столь же неумолимо, как и T1000 из Терминатора-2. Он не остановится до тех пор, пока не поймает игрока. Мы могли бы несколько облегчить его задачу путем добавления некоторой дополнительной логики, способствующей его движению к позиции нанесения удара. Однако, перед тем, как это делать, давайте посмотрим программу, моделирующую преследование. Листинг программы 13.1 рисует две точки: одну голубую (вы), а другую красную (противник). Что бы вы ни предпринимали, красная точка пытается настичь вас. Для движения (или, я бы сказал, бега!) нажмите клавишу U - перемещение вверх, N - вниз, H -влево и J - вправо. Для выхода из программы нажмите Q.

Листинг 13.1. Программа Терминатор (TERM.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ////////////////////////////////
#include <stdio.h>
#include <graph.h>
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ ///////////////////////////////////
// указатель на переменную BIOS, содержащую текущее
// значение системного таймера. Это значение изменяется
// с частотой 18.2 раза в секунду
usigned int far *clock = (unsigned int far *)0x0000046C;
// ФУНКЦИИ //////////////////////////////////////////////////
void Timer(int clicks)
{ // Эта функция использует значение таймера для формирования
// задержки. Необходимое время задержки задается в "тиках"
// интервалах в 1/18.2 сек. Переменная, содержащая 32-битовое
// текущее значение системного таймера, расположена
// по адресу 0000:046Ch
unsigned int now;
// получить текущее время
now = *clock;
// Ничего не делать до тех пор, пока значение таймера не
// увеличится на требуемое количество "тиков".
// Примечание: один "тик" соответствует примерно 55мс.
while(abs(*clock - now) < clicks){}
} // конец функции Timer
// ОСНОВНАЯ ПРОГРАММА ///////////////////////////////////
void main(void)
{
int px=160,py=100, // стартовая позиция игрока
```

```
ex=0,ey=0;     // стартовая позиция врага
done=0;        //флаг окончания работы программы
_setvideomode(_MRES256COLOR);
printf("     Terminator - Q to Quit");
//сновной игровой цикл
while(!done)
{
// удалить точки
_setcolor(0);
_setpixel(px,py) ;
_setpixel(ex,ey) ;
// передвинуть игрока
if (kbhit())
{
// куда движется игрок?
switch(getch())
{
case 'u' : // вверх
{
py-=2;
} break;
case 'n': // вниз
{
py+=2;
} break;
case 'j': // вправо
{
px+=2;
} break;
case 'h': // влево
{
px-=2;
} break;
case 'q':
{
done=1;
}break;
} // конец оператора switch
} // конец обработки нажатия клавиши
// переместить врага
// начало работы "мозга"
if (px>ex) ex++;
if (px<ex) ex--;
if (py>ey) ey++;
if (py<ey) ey--;
// конец работы "мозга"
// нарисовать точки
_setcolor(9);
_setpixel(px,py);
_setcolor(12);
_setpixel(ex,ey);
// немного подождать
Timer(2);
} // конец основного игрового цикла while
_setvideoroode(_DEFAULTMODE) ;
} // конец функции main
```

Листинг 13.1 показывает как просто сконструировать набор действий, кажущихся на вид разумными. Но на самом деле, прежде чем мы получим полноценный искусственный интеллект, эти строки программы должны претерпеть массу усовершенствований.

Уклонение

Пока мы еще не ушли слишком далеко в наших рассуждениях, продолжим нашу Дискуссию разговором о явлении прямо противоположном преследованию — об уклонении. Чтобы сконструировать создание, уклоняющееся от игрока, нам нужно сделать в точности противоположное предыдущим действиям. Алгоритм 13.2 в общих чертах показывает это.

Алгоритм 13.2. Алгоритм Уклонения.

```
//пусть (px,py) - позиция игрока и (ex,ey) - позиция противника
while(игра) {
.....// код программы
// Вначале - горизонтальная составляющая перемещения
if ex>px then ex=ex-1
if ex<px then ex=ex+1
//Теперь - вертикальная составляющая
if ey>py then ey=ey-1
if ey<py then ey=ey+1
.... // код программы
```

В этом месте я еще раз хочу заострить ваше внимание на том, что игры не думают. Персонажи в них только совершают несложные эволюции в пространстве и времени, да определенным образом реагируют на окружающую обстановку и действия игрока. (Некоторые ученые считают, что люди - это не что иное, как набор действий и реакций.)

А теперь перейдем к следующей теме и обсудим такое понятие как «шаблонные мысли».

Шаблонные мысли

На заре создания видеоигр программисты сталкивались с серьезными проблемами, пробуя нанести точки на экран. Они радовались по поводу удачной имитации тактики боя и написания набора алгоритмов, управляющих существами в их мире.

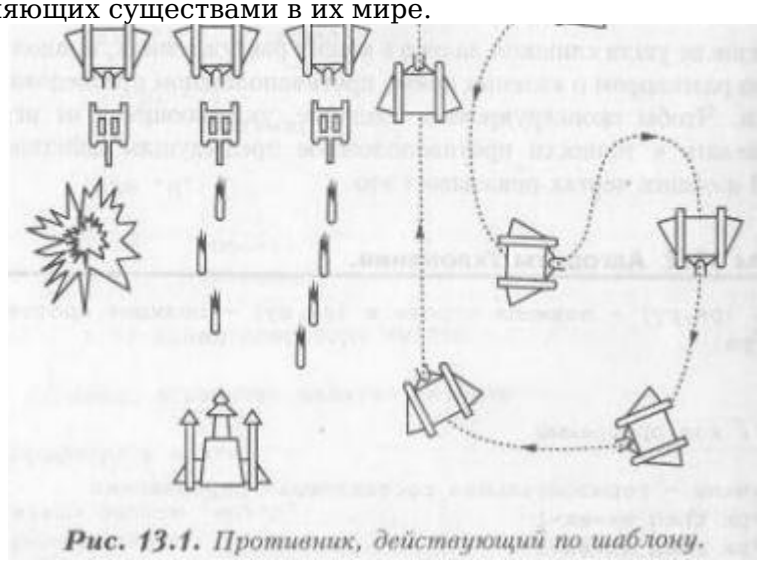


Рис. 13.1. *Противник, действующий по шаблону.*

Увы, они думали, что могут имитировать разумное поведение с помощью шаблонов, то есть описывая рядом цифр траектории перемещения созданий. К примеру, в игре Galaxian маленькие космические корабли делают несколько кругов, стреляют в вас некоторое время, а затем возвращаются на прежнее место в стаю своих сородичей. Посмотрим на рисунок 13.1.

Когда они выполняют эти движения, то всего лишь следуют командам, заранее написанных шаблонов. Конструирование шаблонов - чрезвычайно легкое занятие и они нашли использование во многих сценариях. Как пример, используем шаблоны и случайные числа для моделирования «разума», который можно использовать, например, для управления небольшим космическим кораблем. Он пытается уклониться от вас, и в то же время пробует прицельно стрелять. То, что можно для этого сделать, показано в Алгоритме 13.3.

Алгоритм 13.3. Шаблоны со случайным выбором.

```
// Предположим, что pattern - это массив, содержащий набор команд
// для реализации десяти различных шаблонов поведения
while(идет игра)
{
...код программы
// Проверяем, закончена ли обработка текущего шаблона
if (если обработка команд текущего шаблона закончена)
{
// Выбираем новый шаблон
current_pattern = pattern[rand()%10];
позиция противника = старая позиция +следующий элемент текущего шаблона
Увеличиваем на единицу значение индекса элементов шаблона
...код программы
}
```

Алгоритм 13.3 кажется сложнее предыдущих Алгоритмов Преследования и Уклонения, но на самом деле это не так. В сущности:

- Случайным образом выбирается некоторый шаблон;
- Движение созданий в каждом проходе цикла изменяется в соответствии с направлением, указанным в шаблоне;
- Затем мы переходим к следующему элементу шаблона. Каждый шаблон может включать в себя произвольное количество элементов. Некоторые из них имеют 10 элементов, а иные — 1000.

Важно одно - когда «создание» исчерпывает набор команд, задаваемый одним шаблоном - оно переходит к другому.

Наконец, мы могли бы связать выбор случайного номера шаблона с действиями на основе некой другой логики - например, описываемой Алгоритмом 13.1.

Добавив к Алгоритму 13.3 последний шаг, мы придали ему некоторый аспект, который придал ему большую комплексность. Появилось подобие мыслительного процесса, состоящего из двух стадий. Случайное число подастся на вход селектора шаблонов, который затем выбирает новый шаблон для игры.

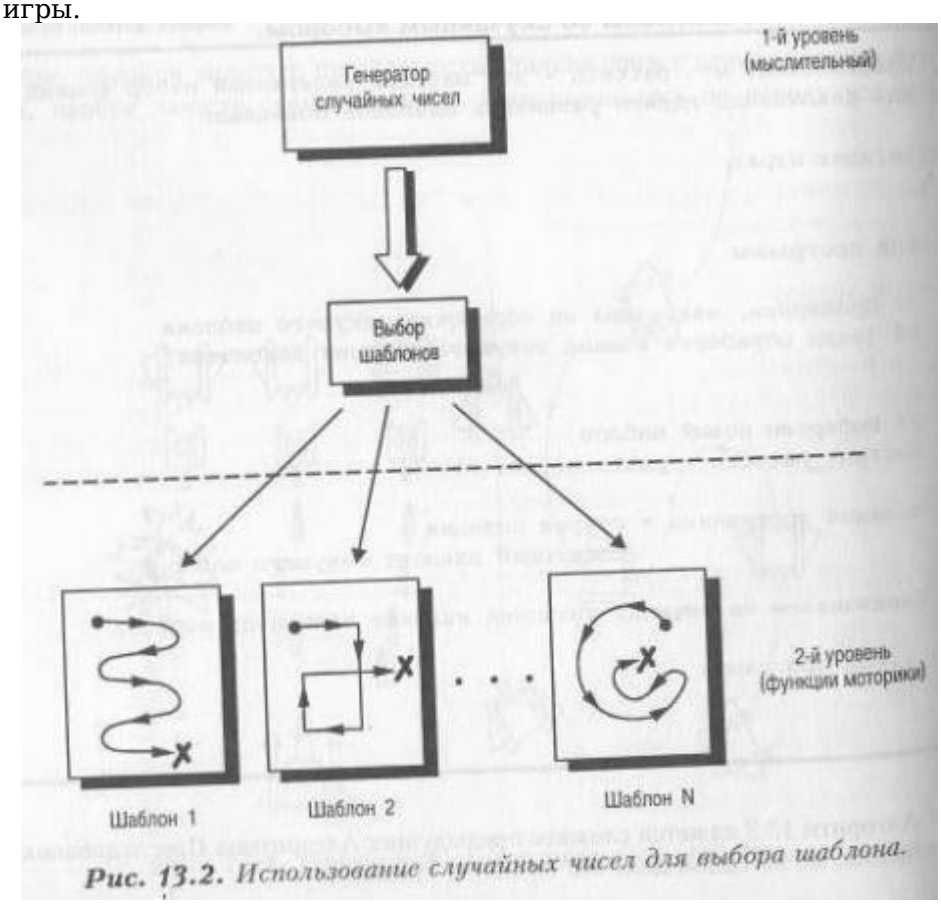


Рис. 13.2. Использование случайных чисел для выбора шаблона.

По аналогии с нервной системой, случайное число можно представить как нервный импульс, а выбранный шаблон действий - как реакцию на него. Посмотрите на рисунок 13.2, чтобы увидеть тот процесс, о котором я говорю.

Для усовершенствования тактики можно добавить измерение расстояния до игрока: если существо находится за пределами некоторого радиуса, используется Алгоритм Преследования, однако, когда оно оказывается достаточно близко, то начинает перемещаться по шаблонным траекториям, выбирая их случайным образом. Эта идея применена в Алгоритме 13.4.

Алгоритм 13.4. Преследование и Танец.

```
while(идет игра)
{
...код программы
if (игрок вне круга с радиусом 50 точек) then преследуем его
else
выбираем случайный шаблон и реализуем его
...код программы
}
```

Для демонстрации использования шаблонов я переработал программу из Листинга 13.1 и привел ее в соответствие с Алгоритмом 13.4. Новая программа показана в Листинге 13.2. Когда враг приближается к игроку, он выбирает один из трех шаблонов и выполняет его до завершения. Далее, в зависимости от расстояния до игрока, противник либо преследует игрока, либо выбирает другой шаблон.

Листинг 13.2. Муха (FLY.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ////////////////////////////////////////
#include <stdio.h>
```

```
#include <graph.h>
#include <math.h>
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ ////////////////////////////////////
//указатель на системную переменную, содержащую
//значение таймера. Содержимое этой 32-битовой ячейки
//обновляется 18.2 раз в секунду
usigned int far *clock=(unsigned int far *)0x0000046C;
// X- и Y-компоненты шаблонов траекторий, по которым будет
// двигаться "муха"
int patterns_x[3] [20]={1,1,1,1,1,2,2,-1,-2,-3,-1, 0,0,1,2,2,-2,-2,-1,0,
0,0,1,2,3,4,5,4,3,2,1,3,3,3,3, 2,1,-2,-2,-1, 0,-l,-2,-3,-3,-2,-2,
0,0,0,0,0,0,1,0,0,0,1,0,1};
int patterns_y[3] [20]={0,0,0,0,-1,-1,-1,-1,-1, 0,0,0,0,0,2,2,2,2,2,2,
1,1,1,1,1,1,2,2,2,2,2,2, 3,3,3,3,3,0,0,0,0, 1,1,1,2,2,-1,-1,-1,-2,-2, -1,-
1,0,0,0,1,1,1,1,1};
//////////////////////////////////////////
void Timer(int clicks)
{
// Эта функция использует значение таймера для формирования
// задержки. Необходимое время задержки задается в "тиках"
// интервалах в 1/18.2 сек. Переменная, содержащая 32-битовое
// текущее значение системного таймера, расположена
// по адресу 0000:0:46Ch
unsigned int now;
// получить текущее время
now = *clock;
// Ничего не делать до тех пор, пока значение таймера не
// увеличится на требуемое количество "тиков".
// Примечание: один "тик" соответствует примерно 55мс.
while(abs(*clock - now) < clicks) {}
} // конец функции Timer
// ОСНОВНАЯ ФУНКЦИЯ ////////////////////////////////////
void main(void)
{
int px=160,py=100,    // начальные координаты игрока
ex=0,ey=0;        // начальные координаты противника
int done=0,           // флаг окончания работы программы
doing_pattern=0,  // флаг выполнения команд шаблона
current_pattern, // номер выполняемого шаблона,
             // принимает значение в интервале 0-2
pattern_element; <// номер выполняемой команды шаблона
_setvideomode(_MRES256COLOR,) ;
printf("             The Fly - Q to Quit");
// главный игровой цикл
while(!done)
{
// очищаем точки
_setcolor(0) ;
_setpixel(px,py);
_setpixel(ex,ey) ;
// Перемещение игрока
if (kbhit())
{
// Определяем направление движения
switch(getch())
{
case 'u': // Вверх
{
py—2;.
} break;
case 'n': // Вниз
{
py+=2;
} break;
case 'j': // Вправо
{
px+=2 ;
```

```
} break;
case 'h': // Влево
{ px-=2 ;
} break;
case 'q':
{
done=1;
} break;
} // конец оператора switch
} // конец обработки нажатия клавиши
// Теперь перемещаем противника
// Начинается работа "мозга"
if (!doing_pattern)
{
if(px>ex) ex++;
if(px<ex) ex--;
if(py>ey) ey++;
if(py<ey} ey--;
// Теперь проверяем, не надо ли начать выполнять
// шаблон. Он начинает выполняться, если игрок
// оказывается в радиусе 50 точек от противника.
if (sqrt(.1+(px-ex)*(px-ex)+(py-ey)*(py-ey))<15)
{
// Даже не думайте использовать функцию sqrt в
// настоящей игре!
// Получаем случайный номер шаблона
curent_pattern = rand()%3;
// Переводим "мозг" в режим действий по шаблону
doing_pattern = 1;
pattern_element=0;
} // конец проверки на попадание игрока
// в "радиус действия"
} // конец действий, для случая, когда шаблон не выполнялся
else {
// перемещаем противника, используя следующий
// элемент текущего шаблона
ex+=patterns_x[current_pattern][pattern_element];
ey+=patterns_y[current_pattern] [pattern_element];
// мы закончили обработку шаблона?
if (++pattern_element==20)
{
pattern_element = 0;
doing_pattern = 0;
} // конец проверки на окончание шаблона
} // конец оператора else
//конец работы, "мозга"
// рисуем точки
_setcolor(9);
_setpixel(px,py);
_setcolor(12);
_setpixel(ex,ey) ; // Немного подождем...
Timer(1);
} // конец цикла while
// восстановление начального видеорежима
_setvideomode(_DEFAULTMODE) ;
}// конец функции main
```

Когда вы запустите программу из Листинга 13.2, то поймете, почему я назвал ее "Муха". Точка, бегающая по экрану и в самом деле напоминает муху. Она приближается к вам и вдруг начинает быстро летать вокруг. И такое поведение персонажа воплощено всего в нескольких строках программы с применением описанных выше алгоритмов. (Гм, так как вы думаете, может быть люди - это и в самом деле комплекс действий и реакций?) Теперь рассмотрим случайные передвижения.

Случайные передвижения

Что я понимаю под случайным передвижением? Это выполнение персонажем какого-нибудь совершенно непредсказуемого действия. Нечто подобное мы уже сделали в программе из Листинга 13.2. Однако, мы можем расширить эту концепцию не только выбирая способ реагирования, но и

определяя, нужна ли реакция как таковая. Почему бы нам не добавить еще один тип поведения нашей "Мухе" - случайное передвижение.

Случайные перемещения используются во многих видеоиграх как один из способов реагирования виртуального мира на действия игрока. Такой образ Действий можно назвать "неявной логикой". Неявная логика помогает принять Рвение при отсутствии части необходимой информации, или вообще информации как таковой.

В видеоиграх мы можем использовать случайные перемещения и неявную логику для выбора траекторий или действий наших созданий.

К примеру, когда вы сталкиваетесь с решением некоторого вопроса, такого как обход препятствия, возникшего на вашем пути, вы попытаетесь избежать встречи с ним, двигаясь направо или налево. Если все варианты равны между собой и одинаково возможны, решение, скорее, случайно и, следовательно является «неявным решением». Если бы кто-нибудь спросил вас, например почему вы решили обойти столб справа, а не слева, то вряд ли вы сумели бы внятно объяснить мотивы своего поведения. В другом случае мы можем использовать случайные переменные для выбора направления движения нашей маленькой «Мухи». Алгоритм 13.5 создает маленькую точку, которой теперь полагается летать, двигаясь в случайном направлении.

Алгоритм 13.5. Случайное передвижение.

```
while(идет игра)
{ ...код программы
if (перемещение точки по текущей траектории закончено) then
{
выбор новой траектории, иными словами, выбираем новый фактор преобразования координат
}
двигаем точку несколько раз по новой траектории
...код программы
}
```

Алгоритм 13.5 моделирует «грубый разум», который выбирает направление дальнейшего движения случайным образом. Напишем программу, которая создает одиноко летящую точку в пространстве с помощью этого алгоритма, Посмотрим, как это работает, на примере программы из Листинга 13.3.

Листинг 13.3. Одинокая муха. (DFLY.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ///////////////////////////////////////
#include <stdio.h>
#include <graph.h>
#include <math.h>
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ //////////////////////////////////////
// Указатель на системную переменную, содержащую
// значение таймера. Содержимое этой 32-битовой ячейки
// обновляется 18.2 раза в секунду
unsigned int far *clock=(unsigned int far *)0x0000046C;
////////////////////////////////////////
void Timer(int clicks)
{
// эта функция использует значение таймера для формирования
// задержки. Необходимое время задержки задается в "тиках"
// интервалах в 1/18.2 сек. Переменная, содержащая 32-битовое
// текущее значение системного таймера расположена
// по адресу 0000:046Ch
unsigned int now;
// получить текущее время
now = *clock;
// Ничего не делать до тех пор пока значение таймера не
// увеличится на требуемое количество "тиков".
// Примечание: один "тик" соответствует примерно 55 мсек.
while(abs(*clock - now) < clicks){}
} // конец функции Timer
// ОСНОВНАЯ ФУНКЦИЯ ////////////////////////////////////
void main(void)
{
int ex=160,ey=100; // начальная позиция "Мухи"
int curr_xv=1,curr_yv=0, // текущие факторы
// преобразования Координат.
clicks=0;             // время, через которое
// "Муха" прекращает движение
//в произвольном направлении.
_setvideomode(_MRES256COLOR) ;
```

```
printf("     The Dumb Fly - Any Key to Quit");
//главный игровой цикл
while(!kbhit())
{
// очищаем точки на экране
_setcolor(0);
_setpixel(ex,ey); // перемещение "Мухи"
// "мозговой штурм"
// закончить ли движение в текущем направлении?
if (++clicks—20)
{
curr_xv = -5 + rand()%10; // от -5 до +5
curr_yv = -5 + rand()%10; // от -5 до +5
clicks=0;
} // конец задания нового направления
// перемещаем "Муху"
ex+=curr_xv;
ey+=curr_yv;
// убеждаемся, что "Муха" находится в пределах экрана
if (ex>319) ex=0; if (ex<0)   ex=319;
if (ey>199) ey=0; if (ey<0)   ey=199;
// конец "мозгового штурма"
// рисуем "Муху"
_setcolor(12);
_setpixel(ex,ey);
// небольшая задержка...
Timer(1);
} // конец цикла while
_setvideomode(_DEFAULTMODE);
} // конец функции main
```

Думаю, после запуска программы вы согласитесь с тем, что мы, наконец получили все необходимое для моделирования летающего «разумного существа». Несколько простых правил, шаблоны и случайные числа помогли нам создать довольно правдоподобную имитацию полета насекомого. Это просто чудесно! Теперь настало время поговорить о конечных автоматах.

Конечные автоматы

Конечные автоматы (КА) - близкие друзья самых разнообразных механизмов. пни царствуют в виртуальном пространстве компьютеров. КА используются в разработке аппаратного и программного обеспечения. Поскольку мы коснулись этой темы, скажу, что КА - это программно моделируемая машина, которая движется от состояния к состоянию, основываясь на входных данных. Кроме того КА обычно запоминает все состояния, в которых он побывал. Посмотрите на рисунок 13.3, на котором изображена диаграмма простейшего КА. (Что именно этот КА делает - не имеет никакого значения).

Как видите, КА имеет набор состояний, представленных кружками и объединенных дугами переходов от состояния к состоянию. Эти дуги показывают правила, по которым происходят переходы из одного состояния в другое. КА могут быть использованы в видеоиграх как один из методов высокоуровневого управления логикой, с помощью которого передаются команды «низ коуровневой» логике. Так же, как наш мозг имеет кору, передающую наши желания в более низкоуровневые моторные доли мозга, КА может быть использован для инициации, управления и перехвата низкоуровневых действий, выполняемых имитатором мозга. Как пример мы могли бы использовать все, что нам уже известно о преследовании, уклонении, шаблонах и случайных передвижениях для констру ирования КА, способного переходить от состояния к состоянию, каждое из которых выполняет одну из перечисленных функций.

Рис. 13.3. *Общая схема КА.*

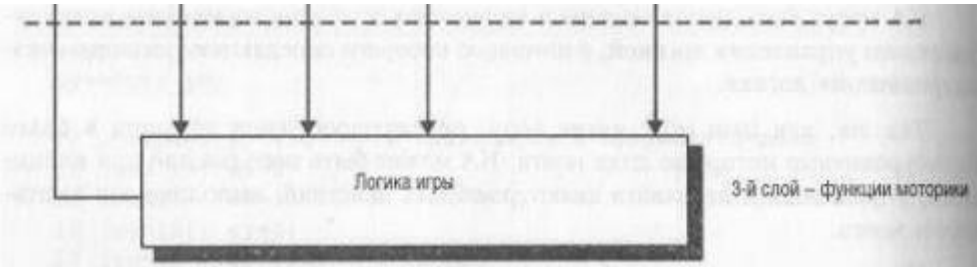


Рис. 13.4. Абстрактный вид КА с точки зрения модели игры.

Теперь у нас есть набор решений, чтобы ответить на вопросы:

- Какая причина заставит КА изменить свое состояние?
- Как заставить КА выбрать следующее состояние?

Это хорошие вопросы - посмотрим, как выглядят видеоигры изнутри. Взгляните на рисунок 13.4. На нем показано абстрактное представление некоторого КА, правда, пока еще без логики смены состояний.

Конечный автомат, управляемый окружающей средой

Теперь наступает приятная часть. Мы можем создать КА, в котором состояний перехода представлены рядом переменных. Другими словами, мы управляем конечным автоматом с помощью самой игровой среды, в которой он находится, так же, как наш собственный мозг реагирует на окружающую обстановку.

Начнем создавать КА с того, что он будет случайным образом выбирать одно из нижеперечисленных состояний:

- преследование;
- уклонение;
- случайное;
- шаблон.

Но в настоящей игре для управления состоянием КА вместо случайных чисел было бы лучше привлечь саму игровую ситуацию. Ниже приведен пример КА действующего по обстоятельствам:

- Если игрок близко, КА переключается в состояние Шаблон;
- Если игрок далеко, можно заставить ПК охотиться за ним, используя состояние Преследование;
- Если игрок обрушил на наше маленькое создание шквальный огонь, КА моментально переходит в состояние Уклонение;
- Наконец, при любой иной ситуации КА отрабатывает состояние Случайное. Рисунок 13.5 демонстрирует подобное поведение конечного автомата.

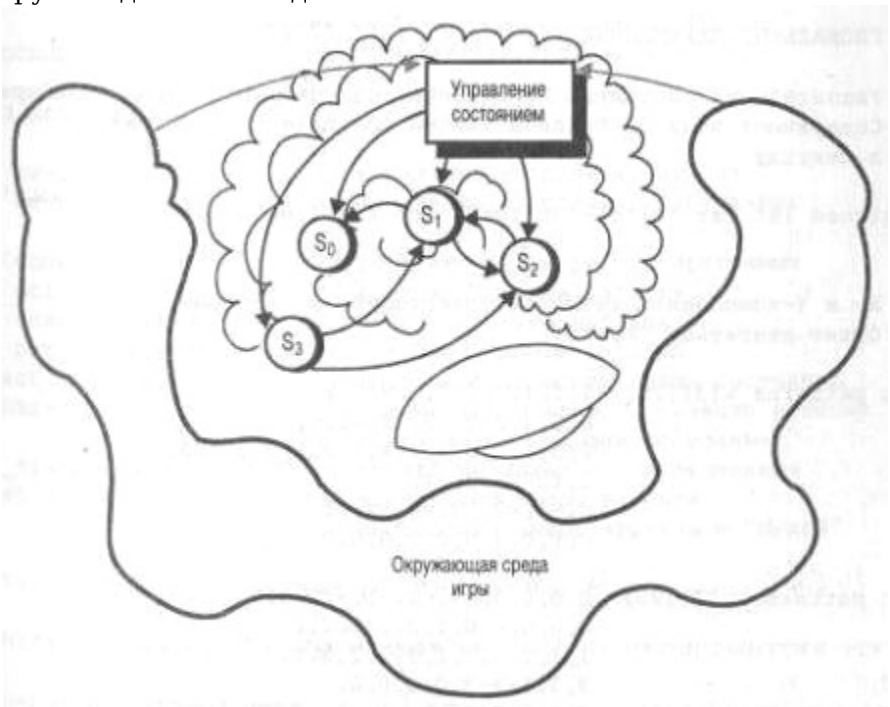


Рис. 13.5. КА, управляемый окружающей средой.

Перейдем к программной реализации описанного КА. Мы не станем создавать сложное игровое пространство. Для демонстрации нам достаточно иметь чистый экран с парой точек,

изображающих игрока и противника. Чтобы понять суть, этого вполне достаточно. Программа из Листинга 13.4 моделирует полет действительно назойливой мухи, которая буквально заедает игрока.

Листинг 13.4. Умная «Муха» (BFLY.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ////////////////////////////////////
#include <stdio.h>
#include <graph.h>
#include <math.h>
// ОПРЕДЕЛЕНИЯ /////////////////////////////
#define STATE_CHASE    1
#define STATE_RANDOM   2
#define STATE_EVADE    3
#define STATE_PATTERN 4
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ ///////////////////////
// Указатель на системную переменную, содержащую значение таймера.
// Содержимое этой 32-битовой ячейки обновляется 18.2 раза
//в секунду
unsigned int far *clock=(unsigned int far *)0x0000046C;
// X- и Y-компоненты шаблонов траекторий, по которым
// будет двигаться "Муха"
int patterns_x[33[20]={1,1,1,1,1,2,2,-1,-2,-3,-1,
0,0,1, 2, 2, -2,-2,-1,0,
0,0,1,2,3,4,5,4,3,2,1,3,3,3,3,
2,1,-2,-2,-1,
0,-1,-2,-3,-3,-2,-2,
0,0,0,0,0,0,1,0,0,0,1,0,1};
int patterns_y[3][20]={0,0,0,0,-l,-l,-l,-l,-l, 0,0,0,0,0,2,2,2,2,2,2,
1,1,1,1,1,1,2,2,2,2,2,2, 3,3,3,3,3,0,0,0,0, 1,1,1,2,2,-1,-1,-1,-2,-2, -1, -1,
0,0,0,1,1,1,1,1};
////////////////////////////////////
void Timer(int clicks)
{
//Эта функция использует значение таймера для формирования
// задержки. Необходимое время задержки задается в "тиках"
// интервалах в 1/18.2 сек. Переменная, содержащая 32-битовое
// текущее значение системного таймера, расположена
// по адресу 0000:046Ch
unsigned int now;
// получить текущее время
now = *clock;
// Ничего не делать до тех пор. пока значение таймера не
// увеличится на требуемое количество "тиков".
// Примечание: один "тик" соответствует примерно 55мс.
while(abs(clock - now) < clicks){}
} // конец функции Timer
// ОСНОВНАЯ ФУНКЦИЯ ////////////////////////////////////
void main(void)
{
int px=160,py=100,       // начальные координаты игрока
ex=0,ey=0;           // начальные координаты противника
int done=0,              // флаг окончания работы программы
doing_pattern=0,     // флаг выполнения шаблона
current_pattern,    // номер текущего шаблона
pattern element,     // текущая команда шаблона
select_state=0,          // флаг необходимости смены состояния
clicks=20,               // количество циклов, в течение которых
// сохраняется текущее состояние
fly_state = STATE CHASE; // "Муха" начинает с преследования
float distance;          // используется при расчете
// расстояния между игроком и "Мухой"
_setvideomode(_MRES256COLOR);
printf("    Brainy Fly - Q to Quit");
//основной игровой цикл
while(!done)
{
```

```
// очистка точек
_setcolor(0);
_setpixel(px,py);
_setpixel(ex,ey);
// перемещение игрока
if (kbhit()) {
// определяем направление движения
switch(getch()}
{
case 'u': // вверх
{
py-=2;
} break;
case 'n': // вниз
{
py+=2;
} break;
case 'j': // вправо
{
px+=2 ;
} break;
case 'h'': // влево
{
px-=2;
} break;
case 'q':
{ done=l ;
} break;
} // конец оператора switch
}
// конец обработки нажатия клавиши
// теперь перемещаем противника
// начинается работа "мозга"
// определяем текущее состояние КА
switch(fly_state)
{
case STATE_CHASE:
{
_settextposition(24,2);
printf("current state:chase   "};.
// реализуем Алгоритм Преследования
if (px>ex) ex++;
if (px<ex) ex--;
if (py>ey) ey++;
if (py<ey) ey--;
//не пора ли перейти к другому состоянию?
if (--clicks==0) select_state=l;
} break;
case STATE_RANDOM:
(
_settextposition(24,2} ;
printf("current state:random   ") ;
// перемещаем "Муху" в случайном направлении
ex+=curr_xv;
ey+=curr_yv;
//не пора ли перейти к другому состоянию?
if (--clicks=0) select_state=l;
} break;
case STATE_EVADE:
{
_settextposition(24,2) ;
printf("current state:evade   ");
// реализуем Алгоритм Уклонения
if (px>ex) ex--;
if (px<ex) ex++;
if (py>ey) ey—;
if (py<ey) ey++;
```

```
//не пора ли перейти к другому состоянию?
if (--clicks==0) select_state=l;
} break;
case STATE_PATTERN:
{
_settextposition(24,2);
printf("current state:pattern   ");
// перемещаем "Муху", используя следующий
// элемент текущего шаблона
ex+=patterns_x[current_pattern][pattern_element];
ey+=patterns_y[current_pattern][pattern_element];
// обработка шаблона закончена?
if (++pattern element==20)
{
pattern_element = 0;
select_state=l;
} // конец проверки завершения шаблона
} break;
default;break;
} // конец обработки текущего состояния
// надо ли менять, состояние?
if (select_state==l)
{
// Выбор нового состояния основан на
// игровой ситуации и неявной логике.
// Для выбора используется расстояние до игрока
distance = sqrt(.5+fabs((px-ex)*(px-ex)+(py-ey)*(py-ey))) ;
if (distance>5&&distance<15&&rand()%2==1)
{
// выбираем новый шаблон
current_pattern = rand()%3;
// переводим "мозг" в состояние действий по шаблону
fly_state = STATE_PATTERN;
pattern_element = 0;
} // конец обработки ситуации "близко к игроку"
else if (distance<10) // слишком близко, убегаем!
clicks=20;
fly_state = STATE_EVADE;
} // конец обработки ситуации "слишком близко"
else
if (distance>25&sdistance<100&&rand()%3==1)
{
// преследуем игрока clicks=15;
fly_state = STATE_CHASE;
} // конец обработки ситуации Преследование
else
if (distance>30&&rand()%2==l)
{
clicks=10;
fly_State = STATE_RANDOM;
curr_xv = -5 + rand()%10; //от -5 до +5
curr_yv = -5 + rand()%10; //от -5 до +5
} // конец "неявной логики"
else
{
clicks=5;
fly_state = STATE_RANDOM;
curr_xv = -5 + rand()%10; //от -5 до +5
curr_yv = -5 + rand()%10; //от -5 до +5
} // конец оператора else // сбрасываем флаг смены состояния
select_state=0;
} // конец Алгоритма Смены Состояния
// Убеждаемся, что "Муха" находится в пределах экрана
if (ex>319) ex=0;
if (ex<0)   ex=319;
if (ey>199) ey=0;
if (ey<0)   ey=199;
```

```
// конец работы "мозга"
_setcolor(12);
_setpixel(ex,ey);
// Немного подождем...
Timer(1);
} // Конец цикла while
_setvideomode(_DEFAULTMODE) ;
} // Конец функции main
```

Запустив программу, вы наверняка обратите внимание на кажущуюся сложность поведения «Мухи». А ведь для получения такого результата использована весьма простая схема! Возможно, и вы заметите то, что увидел я - чуть ли не человеческую способность мыслить.

Управление приоритетным состоянием

Можно еще усовершенствовать наш КА, если ввести управление сменой состояний с помощью некоторых переменных и функций. На практике это означает возможность изменения текущего режима, не дожидаясь полного завершения программы, отвечающей за его выполнение.

В имитаторе движения «Мухи» каждое состояние выполнялось до полного завершения. Но если включить в программу проверку выполнения или невыполнения некоторых условий, это позволило бы КА «выпрыгнуть» из состояния, не дожидаясь окончания его «отработки».

Этим заканчивается наш разговор о конечных автоматах, которые могут быть использованы в наших играх для моделирования поведения существ и придания им видимости наличия интеллекта. Теперь к имеющейся у нас системе высокоуровневого управления не помешает добавить низкоуровневое функционирование.

Вероятностные автоматы

Наверное, вы уже поняли, как вероятность и случайные числа могут быть использованы для выбора направлений и состояний. Мы научились использовать случайные последовательности для конструирования «характера» персонажей. Я имею в виду, что «Муха» в нашем предыдущем примере могла самостоятельно выбирать различные состояния, основываясь на окружающей обстановке. Если несколько изменить метод выбора состояний, основанны на генерации случайных чисел (то есть, создать условия, при которых вход в опредёленное состояние стал бы легче или тяжелее), то, в конечном счете, нам удалось бы изменить "характер" «Мухи».

Скажем, нам захотелось иметь в игре две «мухи». Если одну и ту же программу использовать для создания траектории движения каждой «мухи», они бы действовали одинаково. Во многих случаях большего и не требуется. Однако гораздо интересней иметь много «мух» с небольшими различиями в поведении. Это можно было бы реализовать изменением диапазона случайных чисел во всех строках программы, где выбираются состояния «мухи». Но такой подход будет очень грубым. Мы пойдем другим путем — путем создания общего метода управления характером персонажей, основанного на вероятности.

В искусственном интеллекте «индивидуальность» означает возможность существ по-разному выполнять определенные действия при одних и тех же обстоятельствах. Например, у меня есть несколько достаточно активных друзей, которые захотели бы слегка проучить плута, попытавшегося их надуть. Но у меня также есть друзья, которые более спокойны и предпочитают сначала думать, а потом действовать. Скорее всего, мошеннику удалось бы как-то с ними договориться. То, что мы видим на данном примере и является «индивидуальностью». Каким именно способом это будет достигнуто, не принципиально, важен конечный результат.

В видеоиграх мы могли бы иметь несколько противников, которые постоянно преследуют нас, пока другие в это время неподвижны и стреляют. Третьи трусливы и предпочитают убегать, а не сражаться. Анализируя ситуацию, мы видим, что имеется все тот же набор состояний, но вероятности перехода в них различны для каждого создания. Для моделирования этого процесса мы могли бы сослаться на «таблицу вероятностей» в той части программы, которая выбирает новое состояние. Эта таблица могла бы содержать различные степени вероятности переходов существ в то или иное состояние. К примеру, взгляните на таблицу 13.1, в которой перечислены соотношения вероятностей для трех различных существ в одной игре.

Таблица 13.1. Распределение вероятностей для трех существ в игре.

Состояние Существо 1: Аннигилятор

(А)Атака 50% (П)Преследование 5% (С)Случайное 10%

(Р)Роение 20% (Ш)Шаблон 15%

Состояние Существо 2: Умник

(А)Атака 20% (П)Преследование 30% (С)Случайное

20% (Р)Роение 20% (Ш)Шаблон 10%

Состояние Существо 3: Хныкалка

(А) Атака 5% (П)Преследование 60% (С)Случайное 20% (Р)Роение 10% (Ш)Шаблон 5%

Как вы можете судить по распределению вероятностей, Аннигилятор преимущественно атакует, Хныкалка предпочитает преследование, а Умник -что-то среднее между ними. Прежде чем перейти к разговору о реализации вероятностей, я хочу поговорить еще об одном принципе поведения игровых объектов — о роении. Роение означает, что создания в игре сбиваются в кучи и пытаются как-то сгруппироваться. Это похоже на то, как настоящие солдаты или насекомые иногда стремятся собраться вместе и остаться рядом друг с другом.

В действительности, это не что иное, как выбор общей точки встречи и целеустремленное движение к ней. Мы можем использовать такой тип поведения как совершенно новое состояние, при котором создания кажутся группирующимися.

Моделирование этого состояния не очень сложно:

- Выбрать существа, которые вы хотите сгруппировать вместе;
- Выбрать среднюю относительно всех созданий позицию. Эта точка в пространстве является центром и может быть использована для вычисления траекторий движения всех объектов, которые следуют к ней;
- Вычислить траекторию так, чтобы каждое создание следовало к центру.

В состояние роения имеет смысл переходить, когда слишком много противников вдруг начинают одновременно изрыгать огонь, торпеды и прочие средства уничтожения. Наверное, логичнее всего объединять существ по типам. Посмотрим на рисунок 13.6, на котором показано геометрическое представление роения.

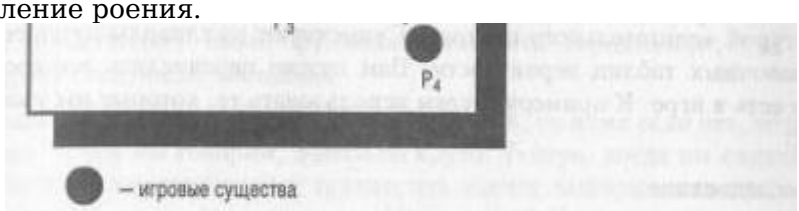


Рис. 13.6. *Геометрическое представление роения.*

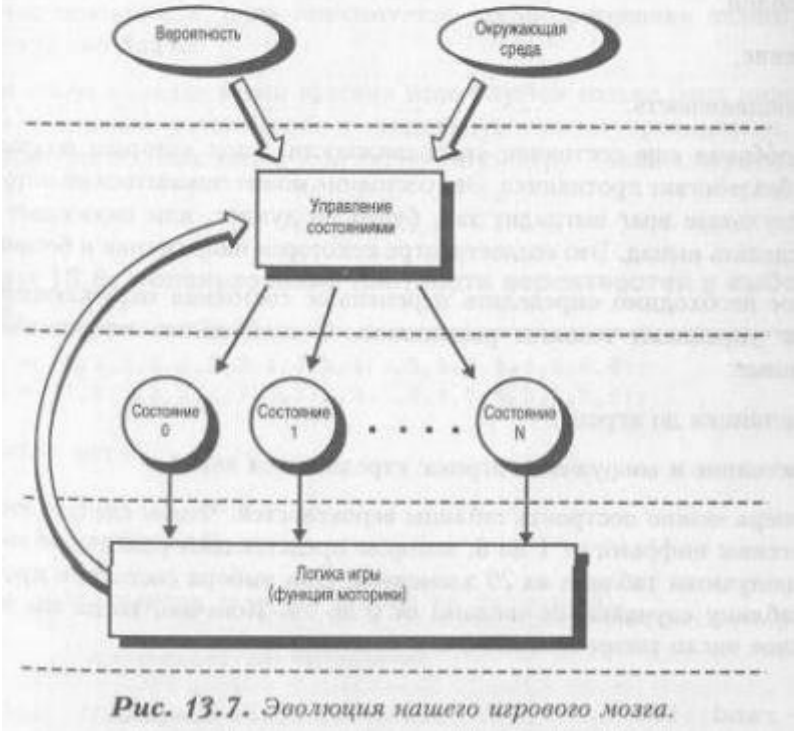


Рис. 13.7. *Эволюция нашего игрового мозга.*

Теперь перейдем к тому, как мы могли бы составить справочные таблицы вероятностей. Давайте посмотрим на наш игровой «мозг», изображенный на рисунке 13.7, с другой стороны.

У нас есть высокоуровневое управление состоянием, которое инициирует новые состояния, основываясь на состоянии окружающей среды и вероятности Это новое состояние является «задействованным вовне» с помощью низкоуровневого управления состоянием, которое, в конечном итоге, является аналогом моторной доли человеческого мозга. Теперь мы двигаемся к тому, чтобы добавить еще один уровень к «мозгу». Этот новый уровень принимает во внимание и состояние окружающей среды, и распределение вероятности. Отдельное распределение вероятности может существовать для каждого создания так что мы можем получить много разных «личностей» с одной и той же общей архитектурой «двигательного центра». Существуют миллиарды путей составления справочных таблиц вероятности. Вам нужно перечислить все состояния которые есть в игре. К примеру, будем использовать те, которые мы уже имеем.

Вот они:

- Преследование;
- Уклонение;
- Случайное;
- Шаблон;
- Роение;
- Неподвижность.

Я добавил еще состояние «неподвижности», под которым подразумеваю полное бездействие противника. Это состояние может показаться не интересным, но в результате враг выглядит так, будто он думает, или поджидает игрока, чтобы сделать выпад. Это создает в игре некоторое напряжение и беспокойство.

Нам необходимо определить переменные состояния окружающей среды, которые управляют нашими решениями. С этой целью можно задать две переменные:

- Дистанция до игрока;
- Состояние и вооружение игрока: стреляет или нет.

Теперь можно построить таблицы вероятностей. Чтобы сделать это, заполним массивы цифрами от 1 до 6, которые представляют различные состояния. У нас получится таблица из 20 элементов. Для выбора состояния проиндексируем таблицу случайными числами от 0 до 19. Конечно, когда мы выбирай случайное число таким способом, как показано здесь:

```
sel= rand()%20
```

числа от 0 до 19 получаются равноценными. Возникает вопрос, как сделать так, чтобы переключение в то или иное состояние происходило чаще или реже? Иначе, что вам делать, чтобы заполнить таблицу тем распределением вероятностей которое вы желаете. К примеру, если вы хотите получить 50% преследования вам необходимо поместить в таблицу 10 единиц. Если роение должно составлять 20%, то вам необходимо занести в таблицу четыре пятерки. Начало вашей таблицы выглядело бы примерно так:

```
int table_1[20] = {1,1,1,1,1,1,1,1,1,1,5,5,5,5,... остальные_состояния}
```

С точки зрения математики мы используем переменную, которая имеет равное распределение вероятности от 0 до 19, чтобы переиндексировать таблицу, которая представляет собой функцию плотности вероятности, где элемент индексирует следующее состояние.

Я надеюсь, что вы понимаете, о чем идет речь, но даже если нет, то поверьте мне - все, о чем мы говорим, довольно круто! Теперь, когда вы видите форму справочных таблиц, мы можем поговорить насчет выбора состояний. Как вы видели в программе «Умная муха» (Листинг 13.4), дистанция до игрока и случайные числа были использованы для выбора следующего состояния. Мы подошли к тому, чтобы сделать для этого же значительно больше. Мы будем учитывать как дистанцию, так и вооружение игрока для выбора одной из двух плотностей вероятности: одна используется, когда противники далеко, а другая - когда они близко.

При таком подходе всеми врагами используется только одна модель «характера», но, внося разнообразие в «индивидуальность» противников, мы без особых дополнительных затрат заметно обогатим игру. Такой алгоритм показан ниже.

Алгоритм 13.6. Использование плотности вероятностей в выборе состояния.

```
table_1 = {1,1,1,2,2,3,3,4,4,4,4,4,5,5,5,5,6,6,6,6};

table_2 = {1,1,2,2,2,2,3,3,3,3,3,3,4,4,5,5,5,5,5,6};

while (идет игра)

{

...код программы

if (выполняются команды одного из состояний)

{

...продолжать их выполнение

} else
```

```
//Выбираем новое состояние
{
if(расстояние между игроком и противником > 100)
{ новое состояние=table_1 [rand()%20];
}
else
{ if(расстояние < 100 и игрок стреляет)
{ новое состояние=table_2[rand()%20];
}
else
// Чисто случайный выбор
{
новое состояние=rand()%7;
}
} // конец оператора else
} // конец выбора нового состояния
} // конец главного цикла while
```

Возможно, некоторые действия ваших созданий, реализующих Алгоритм 13.6, будут выглядеть жутковато, но я гарантирую, что скучно не будет.

Если вы увеличите размеры таблиц вероятностей, то у вас появится возможность получить вполне приличный результат. Далее, вы можете добавить еще несколько операторов IF, усложнить окружающую среду и, наконец, увеличить количество таблиц для достижения большего разнообразия в характерах персонажей вашей игры.

Во всяком случае, пойдем дальше. В следующих разделах мы будем говорить о том, что является частью, как я говорю, «эзотерической коллекции». Это довольно интересно, хотя и совершенно не используются в играх сегодняшнего дня. Быть может, игры завтрашнего дня опять вернутся к этой концепции.

Память и обучение

Если бы мне было нужно определить термины «память» и «обучение», я вероятно сказал бы, что они являются способностью использовать прошлый опыт для решения новых проблем вместе со способностью накапливать и интерпретировать новую информацию.

Вполне возможно наделить создания в наших играх способностью «учиться» и «запоминать», хотя и в достаточно грубой форме. Все, что нам необходим сделать, это ввести несколько переменных и использовать их при выборе состояния и в моторных функциях.

К примеру, мы могли бы позволить каждому существу в нашей игре иметь внутреннюю карту игрового пространства. Если бы существо, войдя в комнату, обнаружило там кучу смертельных врагов, то в следующей раз, когда оно входит комнату, мы могли бы увеличить его агрессивность. Другой пример: если во время боя игрок двигался влево и потом стрелял, игровые существа могли бы запомнить это путем интерполяции траектории игрока в определенном направлении. Любые действия, которые игрок чаще всего предпринимает, могут быть использованы против него, если наделить игровые существа способностью использования этой информации для усовершенствования собственной тактики.

Это только несколько примеров, для чего могут быть использованы память и обучение. Вообще, вполне можно научить игру запоминать информацию о вас в том же объеме, в котором вы запоминаете информацию об игре, а затем использовать накопленный опыт против игрока, хотя это и несколько напоминает самоубийство.

Алгоритмы Поиска. Выслеживание игрока

«Мозги», которые мы сконструировали, получились достаточно умными, так что существо даже получило шанс выжить- И теперь я хочу поговорить о другой довольно простой вещи — как должен быть организован поиск.

Вспомним историю Тезея с Минотавром. Жил когда-то давным-давно Минотавр (человек с головой быка), который преследовал Тезея. В общем, Тезей украл у Минотавра... не помню точно — CD-ROM, или что-то там еще... Тезей попал в ловушку (в действительности, подрядчик, построивший ее, называл ее лабиринтом), но успел выбраться наружу прежде, чем Минотавр его настиг. Вопрос состоит в том, как нужно двигаться по лабиринту, чтобы найти путь к выходу или любой произвольной точке лабиринта, и при этом не застрять в нем навсегда? Давайте взглянем на рисунок 13.8, где изображен пример лабиринта. Существует простой алгоритм, обозначенный здесь как Алгоритм 13.7, который поможет вам найти выход из любого лабиринта. Будьте внимательны, в один прекрасный момент он может вам сильно пригодиться!

Алгоритм 13.7. Путь наружу.

```
do {
```

```
Двигайтесь вдоль правой стены
если встретился проход,
поверните направо
если попали в тупик,
развернитесь на 180 градусов
}пока не выйдете из лабиринта
```

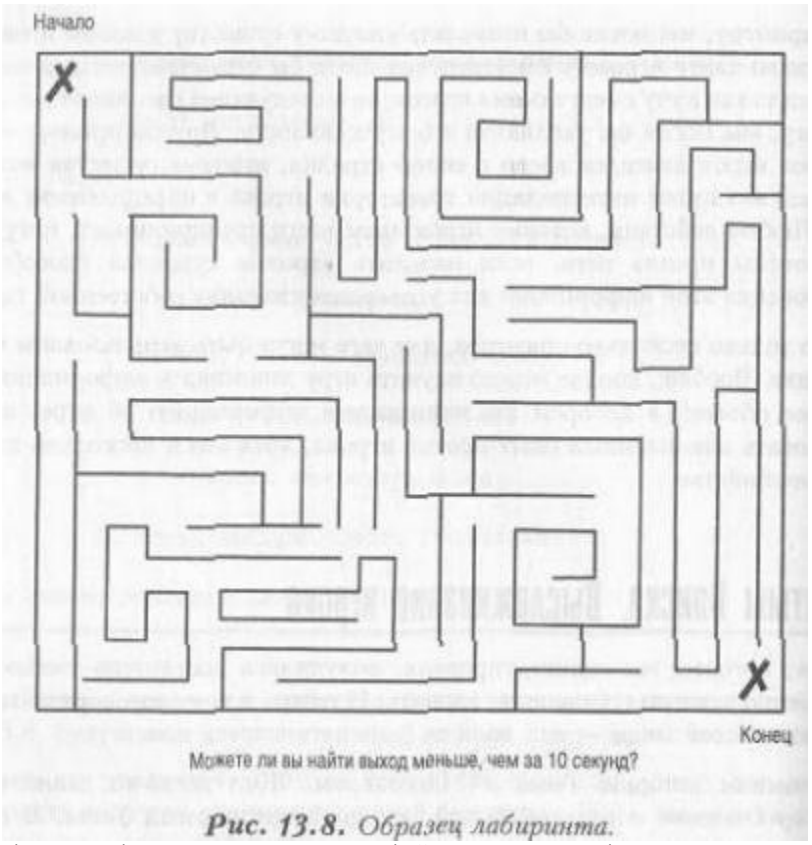


Рис. 13.8. *Образец лабиринта.*

Этот алгоритм работает безупречно: попробуйте его у себя дома. Стартуйте из любого угла вашей квартиры и следуйте алгоритму. Возможно, при известных обстоятельствах вы даже найдете путь к каждой двери.

Так как нас интересуют видеоигры, попытаемся использовать этот алгоритм, чтобы помочь созданиям в игре выследить игрока, не тыкаясь в стены (если мы, конечно, не хотим обманывать игрока, позволив созданиям беспрепятственно проходить сквозь стены). Использование этого алгоритма может привести к затруднениям, если надо обходить объекты круглой формы. Однако мы можем или избегать такой ситуации, или добавить в программу дополнительный алгоритм для предотвращения ходьбы по кругу.

Теория игр

Следующая тема не слишком тесно связана с видеоиграми, а, скорее, относится к стратегическим играм типа шахмат. Тем не менее, я хочу осветить ее, чтобы сообщить вам некоторые интересные идеи.

Теория игр является разделом математики, слишком трудным и слишком абстрактным, для того чтобы применять ее в видеоиграх. Она оперирует набором правил для вычисления оптимального решения или алгоритма, позволяющего игроку победить. Обычно это делается с помощью множества матриц, вероятностей и линейного программирования. Тем не менее, мы можем извлечь из теории игр немало полезного.

Как правило, требуется решить проблему путем оценки текущего положения или игровой ситуации, подразумевая, что игрок точно оценивает свои действия в данной точке в данный момент времени и далее старается улучшить свою позицию. Чтобы выполнить это на компьютере, нам надо:

- Во-первых, мы создаем несколько видов целевых функций, которые могли бы оценивать нашу текущую позицию;
- Далее мы пробуем применить новую тактику или изменить позицию с тем, чтобы посмотреть, улучшит ли это наше положение;
- Если это достигается, мы выполняем это действие;
- Если нет, испытываем следующую тактику.

Именно по такой схеме работают шахматные программы. Компьютер постоянно ведет подсчет своих собственных очков и очков противника, зависящих от текущего расположения фигур, и пытается найти наиболее оптимальный ход. Этот «мыслительный процесс» может в действительности идти на многих уровнях. Компьютер будет проверять каждую из возможностей: если я сделаю так, то мой противник мог бы предпринять то-то..., а если я поступлю эдак, соперник сделает следующее... Так может продолжаться до бесконечности, хотя человек может проиграть даже в том случае, если компьютер будет делать только два шага «проверка-действие».

В видеоиграх эта тактика может быть использована как самое высокоуровневое управление всеми остальными функциями. К примеру, каждую минуту или около этого, компьютер мог бы просчитывать ходы и оценивать, как он действует. Если все его ходы удачны, он мог бы действовать так и дальше. Однако, если какая-то попытка оказалась неудачной, значит, пришло время для

пересмотра тактики.

Итог

В этой главе мы затронули массу интересных тем и даже создали небольшой "мозг", способный имитировать многие функции, присущие живым объектам. Используя комбинации различных алгоритмов, описанных в этой главе, вы сможете создать «виртуальный мозг» достаточно мощный для того, чтобы ваши издания смогли бороться за выживание. Однако, я хочу вам дать пару советов:

- Старайтесь не усложнять «мозг» без необходимости;
- Не пытайтесь сразу создать суперкомпьютер. Вначале пусть ваше первое произведение будет больше похоже на «Денди».

Мы обсудили большинство методов управления поведением, применяемых в сегодняшних видеоиграх. Если вы смогли в них разобраться, вы сможете создавать игры не хуже, а то и лучше чем те, которые существуют сегодня. Индустрия компьютерных игр движется семимильными шагами по пути улучшения графических и звуковых возможностей, часто забывая о такой маленькой детали как интеллект. К счастью, теперь вы имеете возможность привнести «мозги» и стратегию в следующее поколение компьютерных игр.

Связь

Так же, как единственная клетка мозга не могла бы сделать больших успехов в шахматах, играть с компьютером в одиночестве не очень интересно. В связи с последними достижениями в области телекоммуникаций, видеоигры для нескольких игроков становятся все более распространенными. В настоящее время многие программы поддерживают игру двух человек через модем. В этой главе мы начнем изучать системы связи между ПК и приемы, необходимые для создания видеоигр, рассчитанных на нескольких игроков.

В этой главе будут изучены следующие темы:

- Средства связи видеоигр;
- Последовательный интерфейс ПК;
- Функции поддержки последовательного порта ROM BIOS;
- Соединение через нуль-модем;
- Создание коммуникационных библиотек;
- Стратегия коммуникационных видеоигр;
- Синхронизация вектора состояния;
- Синхронизация состояния порта ввода/вывода;
- Синхронизация по времени;
- Модем;
- Написание игры Net-Tank (Сетевой танк) для двух игроков в замкнутом пространстве.

В этой главе мы сконцентрируем внимание на проблемах дизайна игр для нескольких игроков, по возможности не вдаваясь в управление коммуникациями как таковыми. К сожалению, у нас нет времени на изучение методов осуществления связи через модем в полном объеме. Однако мы научимся управлять связью через последовательный порт с помощью нуль-модема. С этими знаниями, потратив дополнительно несколько ночей, вы научитесь связываться через модем. У вас будут все элементы, достаточные для того, чтобы получить нечто работающее. Мы хотим научиться писать видеоигры для двух и более игроков сидящих за своими компьютерами и играющими одновременно без потери синхронизации и прочих проблем. Поэтому нас больше будет интересовать тактика коммуникации в архитектуре видеоигр, а не физическая коммуникация сама по себе.

Средства связи видеоигр

К этому моменту вы уже должны быть готовы писать видеоигры. Вы от начала и до конца научились тому, что для этого необходимо знать. Однако мы до сих пор упускали один важный момент: как написать игру, чтобы в нее могли играть сразу несколько человек? Это нам покуда совершенно неизвестно.

Существуют видеоигры, поддерживающие модемное соединение, но большинство людей или не пользуются этой возможностью, или чувствуют, что она слишком сложна, чтобы к ней прибегать. В любом случае, вопрос в этой области все еще остается открытым и вам предстоит проделать немало работы, если действительно займетесь программированием игр.

Поскольку нас первоначально интересует написание видеоигр для потребителей, мы будем концентрировать наши усилия на использовании серийных портов как средства коммуникации (а не сетевые коммуникации типа IPX/SPX или NETBIOS). Я выбираю параллельный порт в отличие от использования нескольких Ethernet плат по следующим причинам:

- Во-первых, у каждого компьютера есть свой параллельный порт;
- Во-вторых, у многих людей есть свои модемы, с помощью которых они обладают основным средством для игры вдвоем.

Если мы собираемся писать видеоигры для двух или более игроков, то должны рассчитывать на это с самого начала. Вы не можете написать сложную игру и только потом вдруг решить, что делаете ее как игру для нескольких игроков. Необходимо помнить, что разработка двух независимых игр, которые запускаются и синхронно работают на двух различных ПК, требует приличных трудовых затрат и предварительного планирования.

Последовательный интерфейс ПК

Последовательный порт в ПК сконструирован так, что представляет собой систему охватывающую много различных компонентов. Нам, как программистам не обязательно вдаваться в подробности его устройства с точки зрения аппаратуры. Однако полезно иметь представление, как он работает, даже если это никогда не понадобится.

Перед тем как мы погрузимся в описание системы последовательных коммуникаций ПК, давайте рассмотрим основу тех ее возможностей, которые интересуют нас как разработчиков видеоигр. ПК могут поддерживать до семи последовательных портов, несмотря на то, что большинство ПК имеют всего очин или два последовательных порта. Мы можем сконфигурировать их для связи так, что скорость передачи может достигать 115200 бод. Мы можем также выбрать тип четности, количество стоп-битов и битов данных, а также типы прерываний, которые мы хотим

задействовать. Если вы не знаете, о чем идет речь, то задержитесь здесь чуть подольше и постарайтесь понять, о чем я буду говорить.

Прежде всего, нужно сконфигурировать порт, через который мы сможем общаться с другими компьютерами или устройствами, читая или записывая данные в последовательный порт. Аппаратное обеспечение ПК заботится о многих деталях передачи и приема данных. Все, что нам необходимо сделать, это послать в последовательный порт символ для передачи или обработать символ, который вы ожидаете. Для видеоигр важно, что мы посылаем пакеты информации, которые описывают состояние игры на другой машине и наоборот. Эти пакеты состоят из стандартных символов длиной 8 бит.

Единственное, что нам нужно знать, это как открыть последовательный порт, записать в него символы и прочитать ответную информацию. Перейдем ближе к делу и посмотрим, что ПК могут для этого предложить (между прочим, значение некоторых терминов объяснялось чуть ранее).

Универсальный асинхронный приемопередатчик

ПК оборудованы универсальным асинхронным приемопередатчиком (UART) - чипом, который принимает и передает последовательные данные. Существуют Два наиболее популярных UART для ПК:

- Модель 8250;
- Модель 16550.

Можете считать, что они полностью совместимы друг с другом и нам не нужно выяснять, какой из них используется. Единственным их важным отличием является только то, что модель 16550 имеет внутренний FIFO (First In, First Out - "первый вошел - первый вышел") буфер, который располагает входящие - данные так, что они не могут потеряться вследствие задержки обработки. Теперь взглянем на каждый из регистров UART и на то, как получить к ним доступ. После того как мы обязались написать полную библиотеку для связи, необходимо уяснить, как открыть последовательный порт, а также как осуществлять чтение и запись. Написав однажды,соответствующие функции, мы можем сконцентрироваться на целях игры.

Установки и статус UART

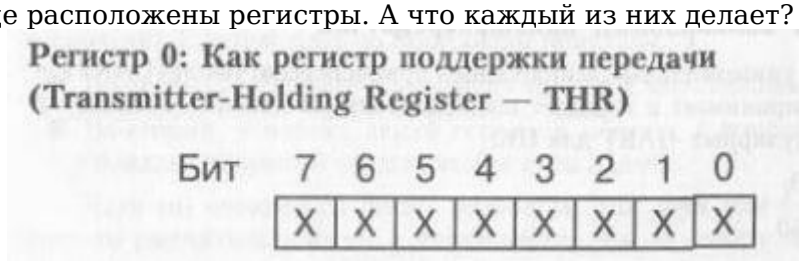
Установки и статус UART управляются через набор внутренних регистров доступных как порты ввода/вывода, адреса которых начинаются от некоторого базового адреса. Базовый адрес определяется номером последовательного порта через который вы хотите связаться. Рассмотрим таблицу 14.1, в которой указаны базовые адреса управляющих регистров UART.

Таблица 14.1. Базовые адреса управляющего регистра UART.

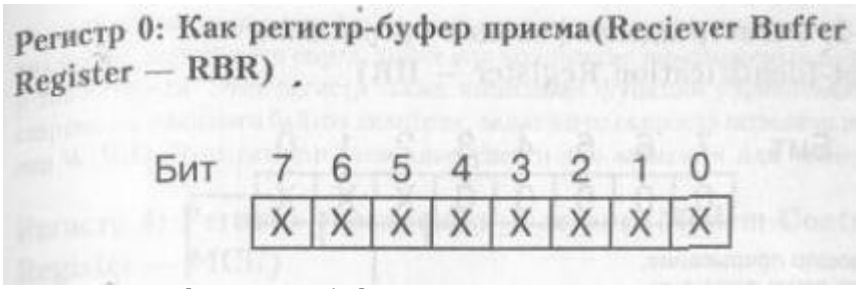
Последовательный порт	Базовый адрес порта
COM1	3F8h
COM2	2F8h
COM3	3E8h
COM4	2E8h

Как видите, если мы хотим играть через последовательный порт COM1, нам необходимо использовать порт 3F8h в качестве базового адреса ввода/вывода. Каждый порт имеет девять регистров, в которые можно писать или из которых можно считывать информацию в зависимости от их типа. Следовательно, для доступа к регистру 1 порта COM1 необходимо использовать адрес ввода/вывода 3F8h+1, то есть 3F9n.

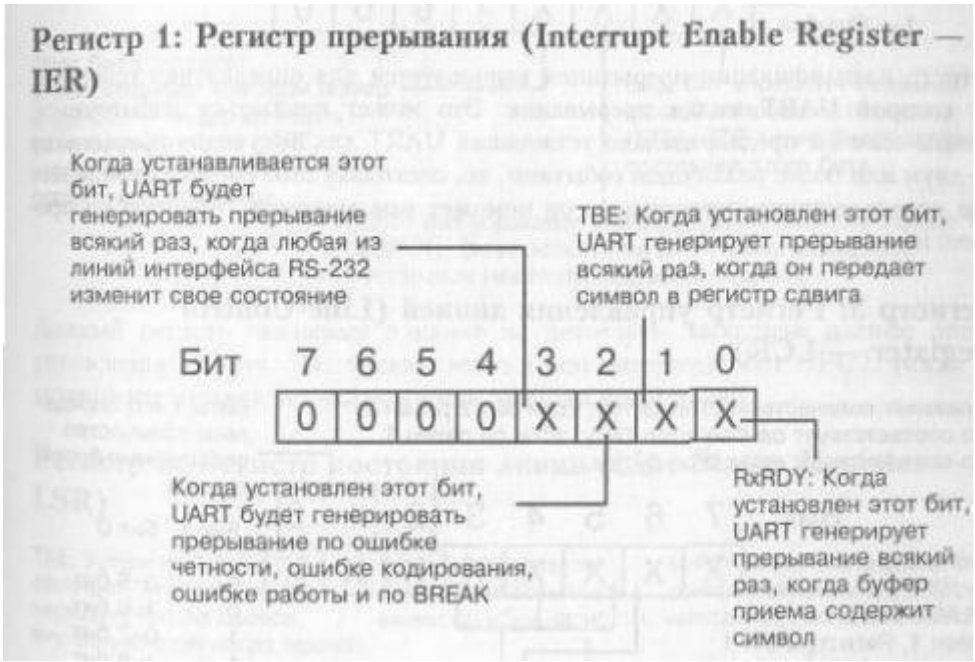
Теперь мы знаем, где расположены регистры. А что каждый из них делает?



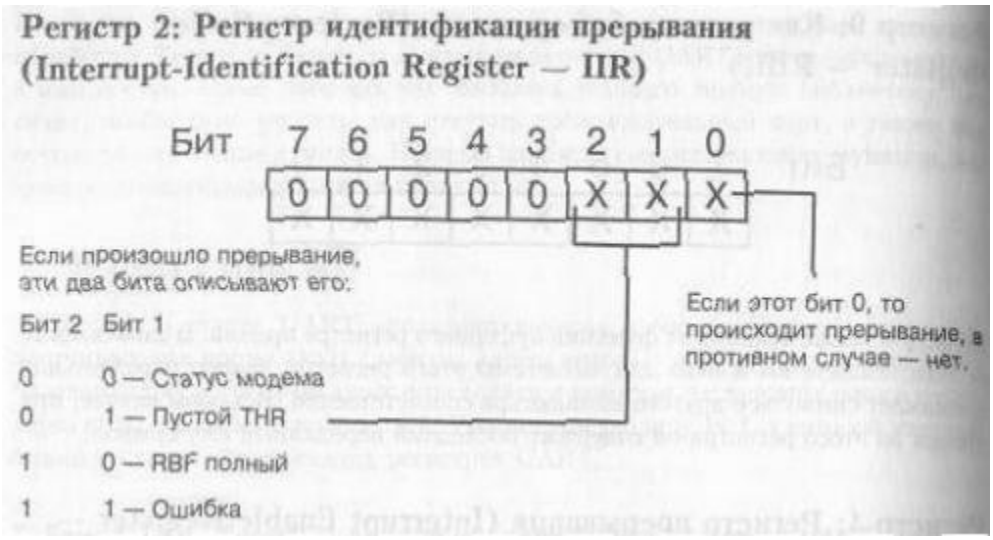
Это регистр поддержки передачи, куда помещается следующий символ для передачи. Если это одиночный байт и вы используете схему передачи, имеющую менее 8 бит, то данные игнорируются, и не передаются вообще.



Регистр 0 также выполняет функции буферного регистра приема. В зависимости от того, пишете вы в него или читаете из этого регистра, буфер передает или принимает символы с другого компьютера соответственно. В любом случае, при чтении из этого регистра он содержит последний переданный ему символ.



Этот регистр используется, чтобы задействовать тот тип прерываний, который может сгенерировать UART. Он доступен как для чтения, так и для записи. После установки серийного порта было бы неудобно постоянно опрашивать его, поэтому для получения входных данных лучше написать процедуру обслуживания прерывания (ISR), которая будет вызываться каждый раз при получении символа. Этот регистр позволяет нам сообщить UART'у, какие именно события Должны вызывать прерывание. Для нас представляет интерес только прерывание RxRDY, которое генерируется при получении символа UART'ом.



Регистр идентификации прерывания используется для определения причины по которой UART выдал прерывание. Это может показаться избыточным однако если вы предварительно установили UART для получения прерывания по двум или более различным событиям, то, поскольку этот регистр определяет тип произошедшего прерывания, он поможет вам выяснить, что именно произошло.

Регистр управления линией используется для изменения некоторых характеристик последовательного порта, таких как количество передаваемых битов данных, тип четности. Этот регистр также выполняет функции управления загрузкой старшего и младшего байтов делителя, задающего скорость передачи и имеющего WORD. Этот регистр также доступен и для записи, и для чтения.

Данный регистр оказывает влияние на некоторые выходные данные линий управления модема. Нас больше всего в нем интересует бит GP02. Когда он установлен, появляется возможность прихода прерываний.

Регистр состояния линии используется, чтобы обрисовать состояние коммуникационного порта. В этом регистре нас интересует пятый бит (TBE), который используется для определения возможности продолжения передачи символов (THR).

Регистр состояния модема используется, чтобы показать состояние линий управления модемом. Для наших целей этот регистр едва ли понадобится. Однако вы могли бы найти применение индикатору звонка (RI). Вы можете написать программу, которая будет перехватывать звонок и когда вызывается ваш номер, например, сообщать об этом соответствующей надписью на экране и звуком.

Регистр 7: Регистр временного заполнения (Scratch-Pad Register)

Не используется

Регистр 8: Менее значимый ключ делителя скорости передачи (Baud-Rate Divisor Latch Least-Significant Byte - DLL)

Предназначен для хранения младшего байта делителя, используемого при вычислении действительной скорости передачи через порт. Окончательная скорость вычисляется так: берут младший и старший банты и используют их как делитель числа 115200. В результате получится скорость передачи. Этот регистр доступен через регистр 0 при установленном 7-м бите (DLAB) регистра 3 (LCR).

Регистр 9: Регистр более значимого байта ключа делителя скорости передачи,(Baud-Rate Divisor Latch Most-Significant Byte-DLM)

Этот регистр используется для поддержки старшего байта делителя, используемого для вычисления действительной скорости передачи через последовательный порт. Окончательная скорость передачи вычисляется следующим образом: берут старший и младший байты и используют их как делитель, на который нужно разделить число 115200. Это дает скорость передачи. Данный регистр доступен через регистр 1 при установленном 7-м бите (DLAB) регистра 3 (LCR).

Аппаратное обеспечение UART

Разобравшись с программным обеспечением UART, давайте взглянем на его аппаратную поддержку. Нас интересуют только две вещи: куда воткнуть кабель и как сделать разъем.

ПК могут иметь два типа последовательных портов:

- 9-штырьковый (разъем типа DB-9);
- 25-штырьковый (разъем типа DB-25).

В таблице 14.2 приведена их распайка.

Таблица 14.2. Распайка для последовательных портов ПК.

Провод	Функция	Обозначение
9-штырьковый разъем		
1	Сигнал наличия несущей	CD
2	Прием данных	RXD
3	Передача данных	TXD
4	Сигнал готовности ввода данных	BTR
5	Земля	GND

6	Сигнал готовности набора данных	DSR
7	Запрос на пересылку	RTS
8	Сигнал очистки для пересылки	CTS
9	Индикатор звонка	RI

25-Штырьковый разъем

2	Передача данных	TXD
3	Прием данных	RXD
4	Запрос на пересылку	RTS
5	Сигнал очистки для пересылки	CTS
6	Сигнал готовности набора данных	DSR
7	Земля	GND
8	Сигнал наличия несущей	CD
20	Сигнал готовности ввода данных	DTR
22	Индикатор звонка	RI

Операции с ROM BIOS

Прежде чем мы начнем писать собственную коммуникационную программу, давайте посмотрим, чем нам может помочь ROM BIOS? Она поддерживает коммуникации через последовательный порт, хотя и весьма ограниченно. Эти функции доступны через прерывание 14h. Существуют функции для открытия последовательного порта, его конфигурирования, чтения и записи символа. Однако существует одна маленькая проблема: эти функции не работают (трудно в это поверить, но они действительно не работают). На самом деле они работают, правда, при особых условиях, которые нам вряд ли удастся создать. К сожалению, отсутствует крайне необходимая нам поддержка ввода/вывода, управляемого прерыванием. В видеоиграх мы должны иметь систему управления событиями, основанную на прерываниях. Из-за этого, а также из-за того, что функции ROM BIOS работают очень медленно, мы можем вообще не рассматривать их как реальное средство передачи битов по проводам.

Теперь, когда я вам сказал об этом, давайте посмотрим, каким Образом все-таки можно соединить два компьютера и создать маленькую сетевую игру.

Соединение через нуль-модем

Как я уже сказал в самом начале, у нас не хватит времени вникать во все тонкости использования модема. Существует слишком много тем, которых мы слегка коснулись, и все они имеет отношение к дизайну видеоигр. Что я намерен сделать вместо этого, так это создать коммуникационную систему, использующую соединение типа нуль-модем, в котором модем, как таковой, отсутствует. *Нуль-модем* — простое соединение, которое связывает два компьютера через последовательные порты. Рисунок 14,1 демонстрирует такое соединение.

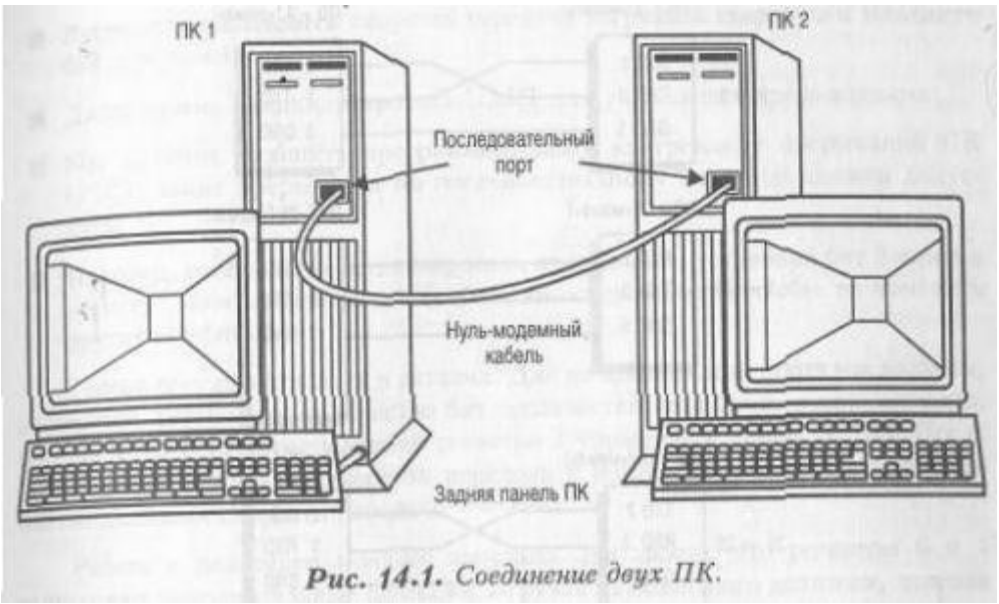


Рис. 14.1. *Соединение двух ПК.*

Чтобы изготовить такое соединение, мы должны взять нуль-модемный кабель с надлежащими разъемами. Это может оказаться не слишком просто, но если вы будете осторожны, то заработает с первой попытки. Мы будем использовать только три типа разъема:

- Линию передачи данных;
- Линию приема данных;
- Землю.

На рисунке 14.2 показано, как сделать нуль-модемный кабель для разных типов разъемов.

Если вы не хотите изготавливать нуль-модемный кабель, вы можете заплатить за него в любом из компьютерных магазинов. (Пожалуйста, не платите больше 15$. Я не могу спокойно смотреть, как люди платят за кусок провода и пластик даже 1.50$). Сейчас официальный нуль-модемный

кабель использует более трех шин, которые я перечислил. Он задействует все шины, имитируя тем самым модемное соединение. Пока у нас есть TXD, RXD и земля, мы тоже при деле. (Дополнительные линии используются для аппаратного обеспечения «рукопожатия», но мы не будем ими пользоваться.)

Отлично, теперь у нас есть нуль-модемный кабель, и мы знаем, какие Регистры что делают в UART'е. Я думаю, что теперь самое время начать писать коммуникационное программное обеспечение.

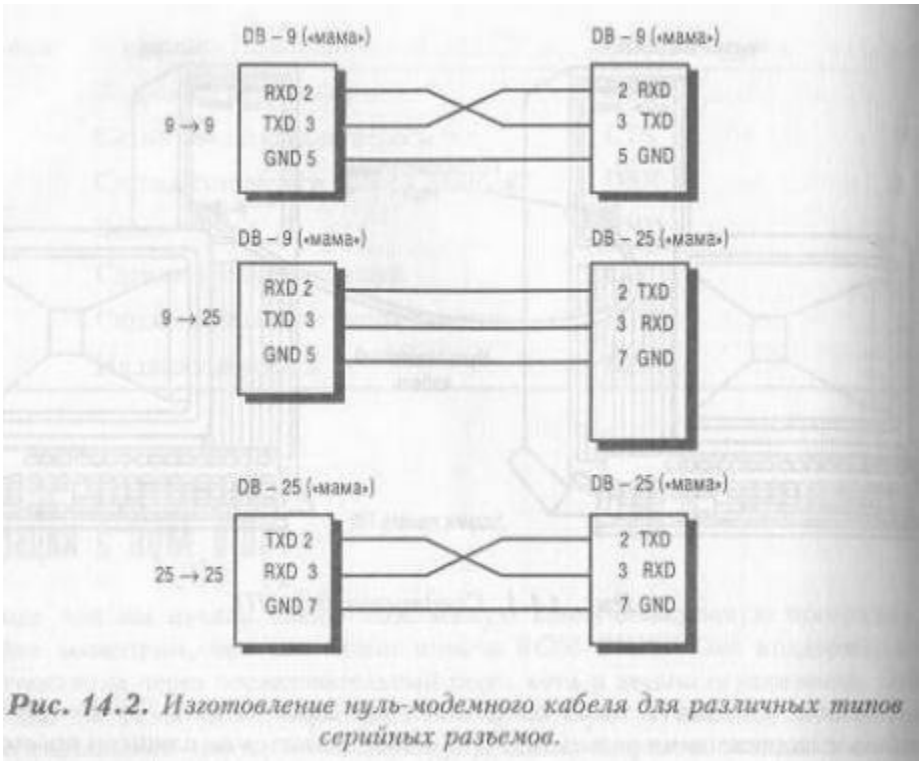


Рис. 14.2. Изготовление нуль-модемного кабеля для различных типов серийных разъемов.

Построение коммуникационной библиотеки

Нам не надо иметь слишком много функций в нашей библиотеке. Фактически, нам достаточно шести функций:

- Для инициализации последовательного порта;
- Для установки процедуры обработки прерывания;
- Для чтения символа из последовательного порта;
- Для записи символа в последовательный порт;
- Для определения состояния порта;
- Для закрытия последовательного порта.

Инициализация последовательного порта

Вот шаги, которым мы должны следовать:

- Сначала нужно установить номера передаваемых битов, количество стоп битов и тип четности. Это делается с помощью регистра управления линии (LCR);
- Затем устанавливается скорость передачи загрузкой старшего и младшего байта делителя;
- Далее нужно инициализировать UART для управления прерываниями;
- Мы должны сообщить программируемому контроллеру прерываний ПК (PIC), какие прерывания по последовательному порту он должен допускать;
- Наконец, необходимо активизировать прерывание, установив бит 3 или 4 в регистре маски прерываний (соблюдая осторожность, чтобы не изменить другие его биты).

Теперь обсудим эти шаги в деталях. Для инициализации порта мы должны, во-первых, установить количество бит, количество стоп-битов и тип четности. Это делается программированием регистра 3 управления линии (LCR). После этого нужно установить скорость передачи с помощью старшего и младшего байтов делителя скорости передачи.

Работа с делителем немного запутана. Вы знаете, что регистры 0 и 1 выполняют дополнительные функции загрузки двухбайтного делителя, деление на который числа 115200 дает результат, используемый UART как окончательная скорость передачи. Однако, как мы знаем, регистры 0 и 1 являются соответственно регистром поддержки передачи (THR) и регистром прерывания (IER). Когда бит 7 регистра управления линией установлен в 1, они получают номера 8 и 9, но продолжают адресоваться как 0 и 1. Ясно?

В качестве примера, давайте установим скорость передачи в 9600 бод. Мы могли бы найти число, которое при делении на него значения 115200 давало бы 9600. Это число 12. Далее мы должны запихнуть его в младший и старший байты. В этом случае младший байт будет равен 12, а

старший - 0. Далее нужно установить бит 7 (DLAB) регистра управления линией в 1 и записать младший байт в регистр 0, а старший - в регистр 1, через которые переданные байты и попадут в регистры 8 и 9. После этого необходимо очистить бит 7 (DLAB) регистра управления линии. Это было бы не так уж и плохо, а?

Затем мы инициализируем UART для приема прерываний. Немного поговорим об этом. Когда UART принимает данные, символ будет оставаться в буфере приема только до тех пор, пока не прибудет следующий, после чего вновь пришедший символ заместит старый, независимо от того, был ли он считан. Мы не можем этого допустить, иначе потеряем информацию. Существует два решения этой проблемы:

- Во-первых, мы могли бы в цикле опрашивать приемный буфер регистра, чтобы не потерять никаких данных. Но это было бы бесполезной тратой времени;
- Во-вторых (и это гораздо лучше), можно написать для этой цели процедуру обработки прерывания (или ISR).

Если вы помните, в двенадцатой главе, "Мнимое время, прерывания и мультизадачность", говорилось, что в Си несложно установить новое прерывание, используя ключевое слово _interrupt. Мы напишем процедуру, которая будет активизироваться всякий раз с приходом прерывания. Но как нам сообщить UART, что прерывание обнаружено? Если вы пристально посмотрите на описание регистров, то поймете, что вам необходимо установить бит здесь бит там, и UART будет делать свою работу. Чтобы назначить прерывания, мы должны установить следующие биты в UART:

• Бит 0 (RxRDY) регистра прерывания (IER) должен быть установлен в 1;

• Бит 3 (GP02) регистра управления модемом (MCR) должен быть установлен в 1.

После этого мы уже готовы принимать прерывания, правильно? Ошибаетесь! Нужно сделать еще одну вещь. Мы должны сообщить программируемому контроллеру прерываний (PIC), какие именно прерывания по последовательному порту он должен задействовать. Чтобы выполнить это, необходимо изменить установки в регистре маски прерывания (IMP) PIC'а, который доступен через порт 21h. Таблица 14.3 показывает обозначение битов IMR.

Таблица 14.3. Регистр маски прерывания (IMR) PIC'а.

Бит 0: IRQ0 - используется для таймера
Бит 1: IRQ1 - используется для клавиатуры
Бит 2: IRQ2 – зарезервирован
Бит 3: IRQ3 - COM2 или COM4
Бит 4: IRQ4 - COM1 или COM3
Бит 5: IRQ5 - жесткий диск
Бит 6: IRQ6 - гибкий диск
Бит 7: IRQ7 - принтер

Таким образом, последняя вещь, которую нам необходимо сделать для обработки прерываний и запуска - активировать нужное прерывание по биту 3 или 4. Но будьте осторожны! Регистр инвертирован, так что 0 означает включен, а 1 — выключен.

Осторожно

Будьте осторожны, когда используете регистр маски прерываний. Он действительно может причинить неприятности компьютеру. Я предлагаю сперва считывать данные и маскировать их по шаблону, а потом уже писать новые данные.

Установка прерывания

Однажды пройдя через все эти фокусы, чтобы установить простое прерывание, можем окончательно инсталлировать наш собственный вектор ISR, зависящий от COM-порта. Запомните, что порты 3 и 4 используют те же самые прерывания, что и порты 1 и 2 соответственно.

Таблица 14.4. Векторы прерывания последовательного порта.

Вектор	Номер	Адресная функция
0x0B	0x002C-0x002F	RS-232 порт 1
0x0C	0x0030-0x0033	RS-232 порт 2

Все что нам нужно сделать для установки нового ISR, это использовать функцию Си _dos_getvect(), чтобы запомнить прежнее значение вектора, и _dos_setvect(), чтобы инсталлировать наш собственный ISR на место старого. Далее, с приходом прерывания (то есть когда получен символ), будет вызываться наша процедура. Звучит это великолепно, но что она будет делать?

Наш ISR должен выполнять только одну задачу — получить символ из регистра приемного буфера (RBR) и поместить его в программный буфер. Чтобы основная программа могла брать поступающие символы по мере надобности, мы должны буферизировать ввод. С этой мыслью создадим буфер с перезаписью и установим его размер равным 128 байтам, хотя, вообще-то, его длина может быть любой.

Алгоритм буферизации работает так. Полученный из RBR следующие символ помещается в буфер в текущую позицию. Далее текущий индекс буфера инкрементируется. Когда позиция записи в буфере доходит до конца, она перемещается к началу. Как вы понимаете, при этом

данные, которые были записаны ранее, окажутся перекрыты. Надеюсь, что до того, как это произойдет основная программа успеет прочитать символы из буфера и обработать полученные данные. Рисунок 14.3 поясняет принцип работы буфера с перезаписью.

Мы должны обсудить еще одну тонкость, прежде чем закончим разговор об ISR. Непосредственно перед выходом из процедуры обработки прерывания необходимо сообщить PIC'у о ее завершении. Для этого в конец процедуры нужно вставить команду записи в порт 20h значения 20h. Если этого не сделать, произойдет сбой системы. Но это — между прочим, ибо пока вы используете функции Си, об этом не стоит беспокоиться. Вот если бы вы решили писать программы исключительно на ассемблере, то вопрос правильного завершения прерываний оказался бы весьма актуален и мы обсудили бы его более подробно. Но давайте пока остановимся на Си.

Листинг 14.1 показывает операции с ISR.

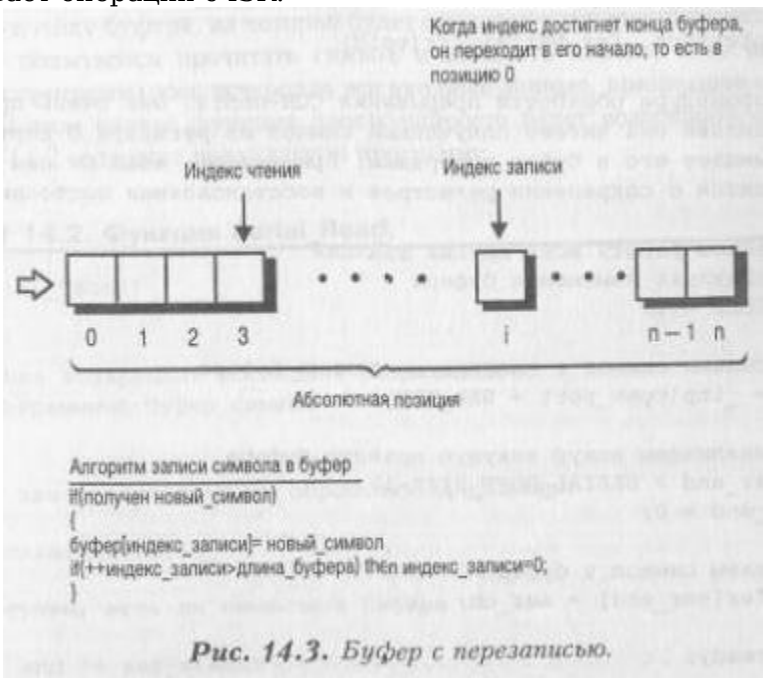


Рис. 14.3. *Буфер с перезаписью.*

Листинг 14.1. Операция ISR.

```
void _interrupt _far Serial_Isr(void)
{
// Это процедура обработки прерывания COM-порта. Она очень проста.
// При вызове она читает полученный символ из регистра 0 порта
//и помещает его в буфер программы. Примечание: язык Си сам
// заботится о сохранении регистров и восстановлении состояния
// запрещаем работу всех других функций
//во избежание изменения буфера
serial_lock = 1;
// записываем символ в следующую позицию буфера
ser_ch = _inp(open_port + SER_RBF);
// устанавливаем новую текущую позицию буфера
if (++ser_end > SERIAL_BUFF_SIZE-1) ser_end = 0;
// помещаем символ в буфер
ser_buffer[ser_end] = ser_ch;
++char_ready;
/ / восстанавливаем состояние контроллера прерываний
_outp(PIC_ICR,0x20);
// разрешаем работу с буфером
serial_lock = 0;
} // конец функции
```

Программа из Листинга 14.1 выполняет все то, о чем мы говорили. Однако стоит обратить внимание на одну маленькую деталь. В программу включена переменная serial_lock, которая оберегает основную программу от конфликт тов связанных с обращением к буферу, пока ISR обновляет его. Такой прием называется «блокировкой» или «семафором». В DOS'е подобной проблемы никогда не возникает по ряду причин, о которых говорить слишком долго. Необходимость регулирования доступа к общим данным возникает только для полностью многозадачных систем. Тем не менее, введение «семафоров» - хорошая практика, даже если на данном этапе такая техника и не нужна. Все, мы почти у цели!

Чтение символа из буфера

Теперь нам необходимо иметь возможность считывать символ из буфера. Это легко. В принципе, достаточно запомнить новый индекс, указывающий на текущую ячейку буфера, из которой будет прочитан следующий символ. Но что если мы попытаемся прочитать символ и изменить индекс, в то время как основная программа уже исчерпала все входные данные, пришедшие от прерывания? В этом случае функция просто-напросто будет возвращать символ 0. Листинг 14.2 содержит подходящую программу.

Листинг 14.2. Функция Serial Read.

```
int Serial_Read()
{
// функция возвращает последний записанный
//в программный буфер символ
int ch;
//ждем завершения функции обработки прерывания
while(serial_lock){}
//проверяем, есть ли символы в буфере
if (ser_end != ser_start)
{
// меняем значение начальной позиции буфера
if (++ser_start > SERIAL_BUFF_SIZE-1) ser_start = 0;
// читаем символ
ch = ser_buffer[ser_start];
// в буфере стало одним символом меньше
if (char_ready > 0) --char_ready;
// возвращаем символ'вызвавшей функции
return(ch) ;
} // конец действий, если буфер не пуст
else
// буфер был пуст - возвращаем 0
return(0);
} // конец функции
```

Функция serial_read получает следующий доступный символ из буфера и возвращает его. Если в буфере не осталось данных, она возвращает 0.

Запись в последовательный порт

Нам осталось сделать функцию, которая могла бы записывать символы в последовательный порт. Для выполнения этого достаточно записать данные в регистр поддержки передачи. Однако мы должны сделать так, чтобы одновременно выполнялись два условия:

- Во-первых, ISR не может быть активен;
- Во-вторых, регистр поддержки передачи должен быть пуст.

Первое условие выполняется как само собой разумеющееся. Однако второе должно быть проверено с помощью пятого бита регистра состояния линии. Если этот бит установлен, передатчик пуст и мы можем послать в порт следующий символ.

Листинг 14.3 содержит программу для передачи символа.

Листинг 14.3. Функция Serial_Write.

```
Serial_Write(char ch)
{
// эта функция записывает символ в буфер последовательного порта,
// но вначале она ожидает, пока он освободится
// примечание: эта функция не связана с прерываниями
// и запрещает их на время работы
// ждем, освобождения буфера
while(!(_inp(open_port + SER_LSR) & 0x20)){}
// запрещаем прерывания
_asm cli
// записываем символ в порт
_outp(open_port + SER_THR, ch);
// разрешаем прерывания снова
_asm sti
} // конец функции
```

Обратите внимание на одну хитрость, примененную в функции Serial_Write: она запрещает прерывания перед записью символа в порт и затем вновь разрешает их.

Коммуникационная программа: NLINK

Программа NLINK завершает наш извилистый путь освоения последовательных коммуникаций для ПК. Я написал эту небольшую коммуникационную программку, чтобы вы могли лучше оценить пройденное. Она соединяет два ПК через COM1 или COM2 и позволяет двум игрокам общаться по нуль-модемному кабелю. Для выхода из программы надо нажать клавишу Esc. Листинг 14.4 содержит законченную коммуникационную библиотеку и главную часть программы NLINK.

Листинг 14.4. Коммуникационная программа NLINK (NLINK.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ////////////////////////////////
#include <dos.h>
#include <bios.h>
#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <graph.h>
// ОПРЕДЕЛЕНИЯ /////////////////////////////////////////
// регистры UART
#define SER_RBF        0    // буфер чтения
#define SER_THR        0    // буфер записи
#define SER_IER        1    // регистр разрешения прерываний
#define SER_IIR        2    // регистр идентификации прерывания
#define SER_LCR        3    // регистр управляющих данных
// и разрешения загрузки делителя
#define SER_MCR        4    // регистр управления модемом
#define SER_LSR        5    // регистр состояния линии
#define SER_MSR        6    // регистр состояния модема
#define SER_DLL        0    // младший байт делителя
#define SER_DLH        1    // старший байт делителя
// битовые маски для управляющих регистров
#define SER_BAUD_1200 96    // значения делителя
// для скоростей 1200-19200 бод
#define SER_BAUD_2400 48
#define SER_BAUD_9600 12
#define SER_BAUD_19200 6
#define SER_GP02       8     // разрешение прерываний
#define COM_1          0x3F8 // базовый адрес регистров COM1
#define COM_2          Ox2F8 // базовый адрес регистров COM2
#define SER_STOP_1      0     //1 стоп-бит на символ
#define SER_STOP_2      4     //2 стоп-бита на символ
#define SER_BITS_5     0     //5 значащих бит на символ
#define SER_BITS 6     1     //6 значащих бит на символ
#define SER_BITS_7     2     //7 значащих бит на символ
#define SER_BITS 8     3     //8 значащих бит на символ
#define SER_PARITY_NONE 0    // нет контроля четности
#define SER_PARITY_ODD   8 // контроль по нечетности
#define SER PARITY EVEN 24    // контроль по четности
#define SER_DIV_LATCH_ON 128 // используется при загрузке делителя
#define PIC_IMR        0x21   // маска для регистра прерываний
#define PIC ICR        0x20   // маска для контроллера
// прерываний (порт 20h)
#define INT_SER_PORT_0 0x0C   // маска для управления
                       // прерываниями COM1 и COM3
#define INT_SER_PORT_1 0x0B   // маска для управления
// прерываниями COM2 и COM4
#define SERIAL_BUFF_SI2E 128 // размер буфера
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ ///////////////////////////////////////
void ( _interrupt _far *01d_Isr) (); // адрес старой подпрограммы
// обслуживания прерываний
// COM-порта
char ser_buffer[SERIAL_BUFF_SIZE];// буфер для приходящих символов
int ser_end = -1,ser_start=-l;      // указатели позиции в буфере
int ser_ch, char_ready=0;           // текущий символ и флаг
// готовности
int old_int_mask;                   // старое значение маски
// контроллера прерываний
int open_port;                      // текущий открытый порт
```

```
int serial_lock = ,0;                // "семафор" для процедуры
// обработки прерывания,
// управляющий записью
// в программный буфер
////////////////////////////////////////////////////////
void _interrupt _far Serial_Isr(void)
(
// это процедура обработки прерывания COM-порта. Она очень проста.
// При вызове она читает полученный символ из- регистра 0 порта
// и помещает его в буфер программы.
// Примечание: язык Си сам заботится о сохранении, регистров
// и восстановлении состояния
// запрещаем работу всех других функций
//во избежание изменения буфера
serial_lock = 1;
// записываем символ в следующую позицию буфера
ser_ch = _inp(open_port + SER_RBF);
//Устанавливаем новую текущую позицию буфера
if (++ser_end > SERIAL_BUFF_SIZE-1) ser_end =0;
// помещаем символ в буфер
ser_buffer[ser_end] = ser_ch;
++char_ready;
// Восстанавливаем состояние контроллера прерываний
_outp(PIC_ICR,Ox20);
// Разрешаем работу с буфером
serial_lock = 0;
} // конец функции
////////////////////////////////////////////////////////
int Ready_Serial()
{
// функция возвращает значение, отличное от нуля,
// если есть в буфере есть символы и 0 - в противном случае
return(char_ready);
} // конец функции
////////////////////////////////////////////////////////
int Serial_Read() {
// функция возвращает последний записанный
//в программный буфер символ
int ch;
// ждем завершения функции обработки прерывания
while(serial_lock){}
// проверяем/ есть ли в символы в буфере
if (ser_end != ser_start)
{
// изменяем значение начальной позиции буфера
if (++ser_start > SERIAL_BUFF_SIZE-1) ser_start = 0;
// читаем символ
ch = ser_buffer[ser_start];
//в буфере стало одним символом меньше
if (char_ready > 0) --char ready;
// возвращаем символ вызвавшей функции
return(ch);
// конец действий, если буфер не пуст
else
// буфер был пуст - возвращаем 0
return(0) ;
}// конец функции ////////////////////////////////
Serial_Write(char ch)
{ // эта функция записывает символ в буфер последовательного порта,
// но вначале она ожидает, пока он освободится.
// Примечание: эта функция не связана с прерываниями-
// и запрещает их на время работы
// ждем освобождения буфера
while(!(_inp(open_port + SER_LSR) 5 0x20)){}
// запрещаем прерывания
_asm cli
// записываем символ в порт
```

```
_outp(open_port + SER_THR, ch);
// снова разрешаем прерывания
_asm sti
} // конец функции ////////////////////////////////////
Open_Serial(int port_base, int baud, int configuration)
{
// Функция открывает последовательный порт, устанавливает его
// конфигурацию и разрешает прерывания при получении символа
// запоминаем базовый адрес порта
open_port = port_base;
// сначала устанавливаем скорость работы
// разрешаем загрузку делителя
_outp(port_base + SER_LCR, SER_DIV_LATCH_ON);
// посылаем младший и старший байты делителя
_outp(port_base + SER_DLL, baud);
_outp(port_base + SER_DLH, 0) ;
// устанавливаем конфигурацию порта
_outp(port_base + SER_LCR, configuration);
// разрешаем прерывания
_outp(port_base + SER_MCR, SER_GP02);
_outp(port_base + SER_IER, 1);
// откладываем, работу с контроллером прерываний,
// пока не установим процедуру обработки прерывания
if (port_base == COM_1)
{
Old_Isr = _dos_getvect(INT_SER_PORT 0);,
_dos_setvect(INT_SER_PORT_0, Serial_Isr) ;
printf("\n0pening Communications Channel Com Port #1...\n");
}
else
{
Old_Isr = _dos_getvect(INT_SER_PORT_1);
_dos_setvect(INT_SER_PORT_1, Serial_Isr) ;
printf("\n0pening Communications Channel Com Port #2...\n");
}
// разрешаем прерывание COM-порта на уровне контроллера прерываний
old_int_mask = _inp(PIC_IMR);
_outp(PIC_lMR, (port_base==COM_l) ? (old_int_mask & OxEF):(old_int_mask & OxF7 ) );
} // конец
функции                                                    ////////////////////////////////
////////////////////////
Close_Serial (int port_base)
{
// функция закрывает COM-порт, запрещает вызов его прерываний
// и восстанавливает старый обработчик прерывания
// запрещаем прерывания по событиям COM-порта
_outp(port_base + SER_MCR, 0) ;
_outp(port_base + SER_IER, 0).;
_outp(PIC_IMR, old_int_mask );
// восстанавливаем старый обработчик прерывания
if (port_base == COM_1)
{
_dos_setvect(INT_SER_PORT_0, 01d_Isr) ;
printf("\nClosing Conuflunications Channel Corn Port #1.\n");
}
else
{
_dos_setvect(INT_SER_PORT_l, 0ld_Isr);
printf("\nClosing Communications Channel Com Port #2.\n");
}
// конец функции // ОСНОВНАЯ ФУНКЦИЯ ////////////////////////////
main ()
{
char ch;
int done=0;
printf("\nNull Modem Terminal Communications Program.\n\n");
// открываем COM1
```

```
Open_Serial(COM_1,SER_BAUD_9600,
SER_PARITY_NONE | SER_BITS_8 | SER_STOP_1);
// главный рабочий цикл
while (!done) {
// работа с символами на локальной машине
if (kbhit()) {
// получаем символ с клавиатуры
ch = getch(); printf("%c",ch);
// посылаем символ на удаленную машину
Serial_Write(ch) ;
// не была ли нажата клавиша ESC? Если да - конец работы
if (ch==27) done=l;
// Если был введен символ "перевод каретки" (CR),
// добавляем символ "новая строка" (LF)
if (ch==13)
{
printf("\n");
Serial_Write(10);
}
}// конец обработки клавиатуры
// пытаемся получить символ с удаленной машины
if (ch = Serial_Read()) printf("%c", ch);
if (ch == 27) { printf("\nRemote Machine Closing Connection.");
done=l;
} // конец обработки нажатия ESC на удаленной машине
} // конец цикла while
// закрываем связь и кончаем работу
Close_Serial(COM_l);
} // конец функции main
```

Изучение принципов коммуникации через последовательный порт ввода/вывода похоже на посещение зубного врача — никому не нравится, но всем приходится через это пройти. Мне жаль, что я подвергаю вас подобной пытке, но это исключительно важно знать. Посему не буду вас дольше истязать и перейду к более интересной теме игровых коммуникаций.

Стратегия игровых коммуникаций

Соединение двух ПК и запуск на них сетевой игры является комплексной задачей, не имеющей какого-то общего решения. Все зависит от конкретной цели, которую вы перед собой поставите и решение проблемы, скорее всего, будет меняться от игры к игре. Однако, существует несколько правил, которые всегда нужно принимать во внимание. Именно об этом мы и поговорим в ближайшее время.

В игре для двух участников, которая запускается на одном компьютере, оба игрока имеют равные шансы влиять на игровую ситуацию. Рисунок 14.4 показывает два различных представления этой разновидности игр.

Однако если такая игра запускается на разных машинах, взаимоотношения игроков с внутренним миром компьютеров представляется более сложным, и это отражено на рисунке 14.5.

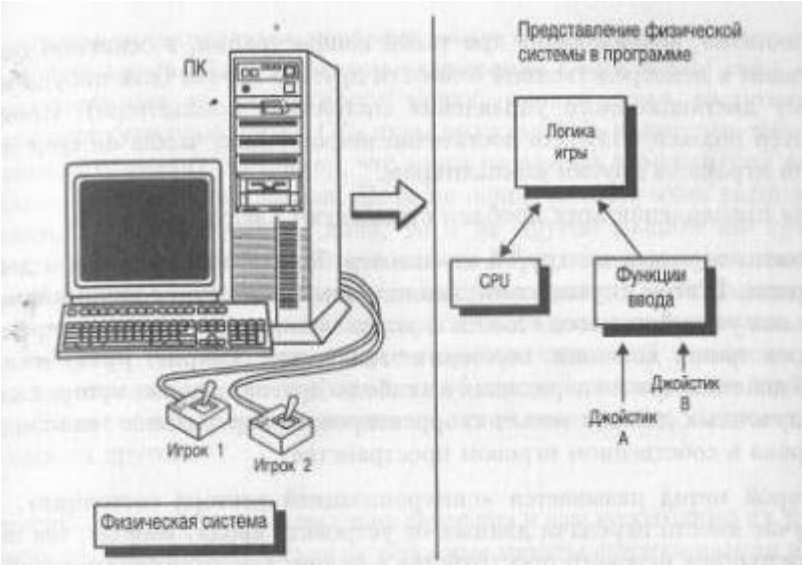


Рис. 14.4. *Различные представления игры для двоих.*

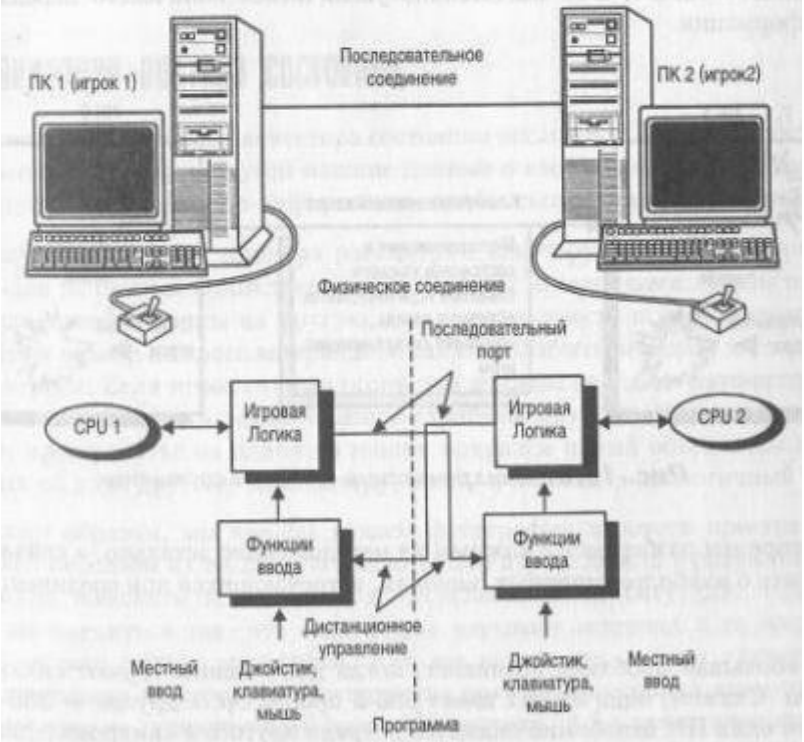


Рис. 14.5. *Игра для двоих на разных ПК.*

Проблемы, возникающие при такой конфигурации, в основном связаны с отсутствием в непосредственной близости другого игрока (как-нибудь мы осветим тему дистанционного управления состоянием компьютера). Кроме того компьютер должен получить достаточно информации, чтобы он смог показать действия игрока за другим компьютером.

Для преодоления этих проблем существуют следующие пути:

- Можно передать на другой компьютер полную информацию о действиях игрока. В этом случае коммуникационный порт будет напоминать виртуальное устройство ввода данных, управляемое другим компьютером. Когда игрок тронет клавиши, переместит мышь или повернет ручку джойстика это действие тут же передается по кабелю другой машине, которая па основе полученных данных может скорректировать виртуальное местоположение игрока в собственном игровом пространстве;
- Второй метод называется «синхронизацией вектора состояния». В этом случае вместо передачи данных от устройств ввода/вывода, мы передаем «состояние» игрового пространства в целом, так что принимающий компьютер может синхронизироваться с передающим, как показано на рисунке 14.6. Этот метод работает достаточно хорошо, однако при его использовании может значительно увеличиться количество передаваемой информации.

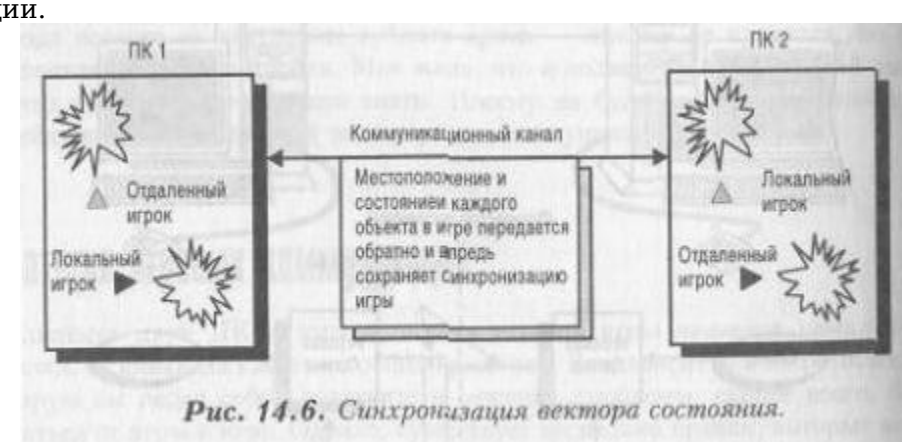


Рис. 14.6. *Синхронизация вектора состояния.*

Вскоре мы разберемся с каждым из методов более, детально, а сейчас стоит поговорить о наиболее типичных ошибках, встречающихся при соединении двух ПК:

- Наибольшая проблема возникает, когда две машины теряют синхронизацию. Скажем, одна из них имеет 586-й процессор, а другая - 386-й. При этом один ПК неизбежно окажется впереди другого и синхронизация будет потеряна. Этот фактор должен быть принят во внимание еще на этапе разработки игры;
- Следующая потенциальная проблема может быть вызвана так называемым «недетерминированным эффектом наложения» (я расскажу лишь о некоторых лежащих иа поверхности вещах, однако этого достаточно для понимания сути проблемы). Обе игры должны быть полностью детерминированы. Это значит, например, что мины на разных компьютерах не могут оказаться в различных местах. Если на одной машине мина расположена скажем, в центре игрового поля, то и на другой машине ей лучше бы оказаться в том же месте. Точно так же, при использовании генератора случайных чисел для управления поведением существ, необходимо, чтобы на обеих машинах генерировалась одна и та же последовательность случайных величин. Единственным путем преодоления этой проблемы может служить передача полной информации об игровой ситуации, так чтобы даже случайные события, происходящие на одной машине, без искажений отражались на другой,

Эти проблемы действительно очень серьезны и вам необходимо их тщательно проработать. Мы кратко обсудили основные методы синхронизации и теперь уже можно поговорить о них более подробно.

Синхронизация вектора состояния

Реализовать синхронизацию вектора состояния несложно. Для этого достаточно непрерывно передавать другой машине данные о состоянии игрового пространства и принимать ответную информацию, чтобы скорректировать обстановку.

Давайте в качестве примера рассмотрим некоторую игру, в которой двое участников на разных компьютерах ведут дуэль с астероидами. Чтобы передать состояние одной машины на другую, мы должны учесть и местоположение, и скорость, и размер каждого астероида, а также не забыть передать и координаты самого игрока. Если игрок открыл огонь, мы должны передать соответствующее сообщение и об этом, а также описать атрибуты оружия. Кроме того, если в игровом пространстве на одной из машин появился новый объект, мы должны сообщить об этом другому компьютеру, чтобы и он создал аналогичный объект.

Таким образом, мы как бы делаем фотографии игрового пространства и постоянно передаем их на другую машину. Это нужно делать в разумном темпе, так, чтобы избежать всевозможных несогласованных ситуаций. Например, может же случиться так, что один игрок взрывает астероид

в то время, как другой на него только нацелился. Как вы понимаете, в этом случае нужно удалить астероид из игрового пространства прежде, чем снаряд второго игрока достигнет уже не существующей цели. На рисунке 14.7 с некоторым преувеличением показано, что может произойти,. если система выйдет из состояния синхронизации.

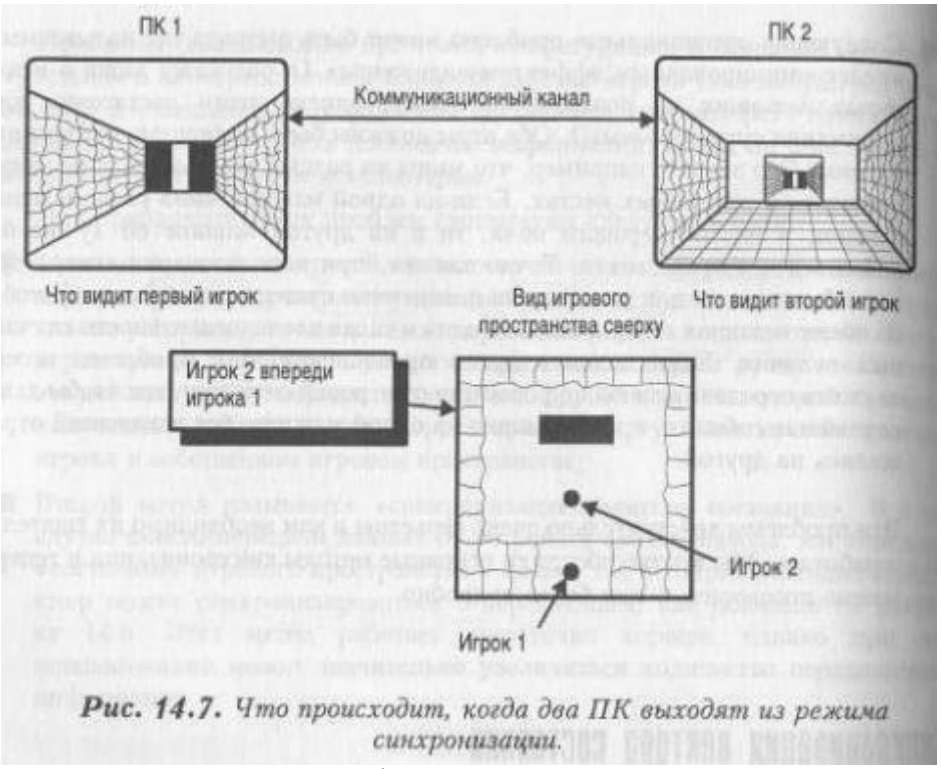


Рис. 14.7. *Что происходит, когда два ПК выходят из режима синхронизации.*

Синхронизация вектора состояния работает прекрасно и совершенно надежно, потому что абсолютно все происходящее на одной машине передается на другую. Однако, как я подозреваю, это не так легко реализовать: ведь во внимание принимаются все возможные характеристики состояния игры, и в результате итоговая информация, передаваемая через коммуникационный канал, оказывается достаточно объемистой.

Следующий метод, о котором мы сейчас поговорим, более легок для понимания и называется синхронизацией состояния ввода/вывода.

Синхронизация состояния ввода/вывода

Синхронизация состояния ввода/вывода является методом, при котором статус устройств ввода данных передается на другой ПК в реальном времени. Все, что игрок делает на одном компьютере, принимающая система воспринимает как входные данные, которые использует для корректировки в своем игрового пространстве поведения образа отдаленного игрока. Рисунок 14.8 пояснее сказанное.

Рис. 14.8. *Передача устройств входных данных на другую машину.*

Этот метод четко работает до тех пор, пока обе системы остаются синхронизированными и не происходит никаких случайных изменений игровой ситуации. Если же подобное произойдет, то другая машина не сможет «узнать» об этом, потому как данный способ не предназначен для передачи такого рода информации.

Если вам все же потребуется, чтобы происходили какие-то случайные события, вы должны воспользоваться первым методом, чтобы сообщить об изменениях другой машине. В дальнейшем мы объединим оба способа синхронизации вместе - это совершенно неизбежно.

Для синхронизации состояния ввода/вывода необходимо;

- Опросить текущее состояние устройств ввода данных, будь то джойстик или клавиатура;

- Объединить их вместе в пакет и послать через коммуникационный канал.

Термин пакет подразумевает объединение разносортной информации. Поэтому для пересылки пакетов мы должны принять ряд соглашений, чтобы последовательные коммуникационные системы «знали», что означает та или иная часть информации. Скажем, мы решили передать через коммуникационный канал положение ручки джойстика одновременно с состоянием его кнопок.

Формат пакета для передачи этих данных мог бы выглядеть примерно так, как это Показано в таблице 14.5.

Таблица 14.5. Образец пакета информационного пространства ввода/вывода.

№ байта	Обозначение	Смысл
0	J	Установленное состояние джойстика
1	data_x	Байт X-координаты джойстика
2	data_y	Байт Y-координаты джойстика
3	buttons	Байт состояния кнопок
4	(period)	Конец передачи

Составить такой пакет чрезвычайно просто. Не сложно и заставить программу передавать его на другую машину с частотой, необходимой для сохранения синхронизации. В то же самое время, другая машина могла бы предоставить в распоряжение первой собственный пакет состояния джойстика.

Здесь есть один не вполне очевидный нюанс. Дело в том, что оба компьютера в равной мере думают друг о друге как об удаленном игроке. (Есть в этом что-то от фантастических романов, описывающих путешествия в параллельные миры и во времени, - частенько путаешься и начинает болеть голова.)

Это все, что я хотел сказать по поводу синхронизации состояния ввода/вывода. Мы еще продолжим разговор о сохранении режима синхронизации, но сейчас нам нужно обсудить такую малоизученную область, как временная синхронизация.

Временная синхронизация

Временная синхронизация означает привязку обеих машин к некоему общему таймеру, который может быть как внутренним, так и внешним. Как я уже говорил, необходимо, чтобы обе игры протекали в одном и том же темпе и каждое событие происходило на обоих компьютерах одновременно. Существует много путей для этого, но здесь приводятся только два из них:

- Один путь состоит в обмене данными между компьютерами с определенный интервалом времени, который выбирается одинаковым для обеих машин. К примеру, машины производят обмен каждые 1/30 секунды. В результате система будет терять синхронизацию не более чем на 1/30 секунды.
- Другой технический прием основан на ожидании посылающим компьютером подтверждения того, что сообщение принимающим ПК получено. Это показано на рисунке 14.9.

Первый компьютер передает свое состояние другому. .Как только вторая машина подтвердит получение этого сообщения и передаст свое состояние, ПК1 может продолжать работу. И запомните, что с точки зрения второго ПК все должно происходить наоборот. То есть после соединения компьютеров один из них получит сообщение первым и тут же вернет его назад - сообщение начнет ходить взад и вперед.

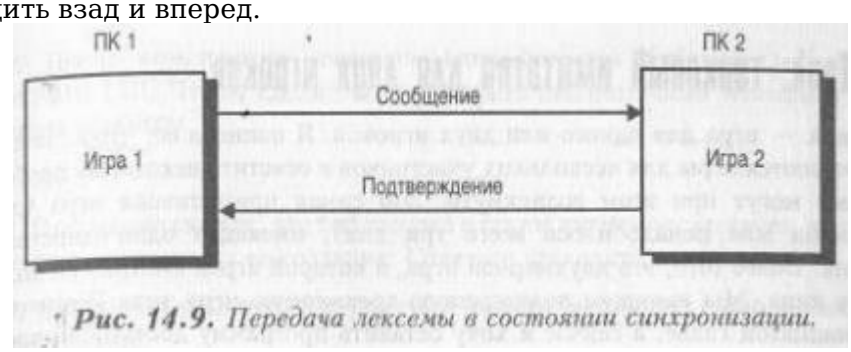


Рис. 14.9. Передача лексемы в состоянии синхронизации.

Этот метод имеет интересный побочный эффект: для сохранения состояния синхронизации компьютеры вынуждены работать с более низкой скоростью. Это происходит оттого, что максимальная задержка определяется не быстродействием ПК, а тем, как долго каждой из машин обрабатываются сообщения.

Прекрасно, теперь вы просто эксперты по коммуникациям. Это было не так уж и сложно, не правда ли? Даже если вы и не стали специалистом, то, по крайней мере, должны неплохо разбираться в этом вопросе и у вас появилось несколько технических приемов для решения возможных проблем. Перед тем как мы приступим к игре Net-Tank, я хочу сказать пару слов о модеме.

Модем

Соединение через модем не так сложно, но описание работы с модемом добавило бы еще немало страниц, на которые у нас просто нет времени. Модем управляется по тем же принципам, что и последовательный порт, с которым он соединен. Единственное отличие состоит в том, что модем прослушивает последовательный порт, и если слышит специальную последовательность символов, то думает, что вы разговариваете с ним, и будет интерпретировать команды и выполнять их. Эти команды называются набором AT-команд. Они поддерживают все функциоальные возможности, которые вам нужны для дозвона, ответа и осуществления телефонного соединения.

Как только соединение произошло, это стало прозрачно для вас. Вы можете забыть, что соединились через модем. Ну ладно, теперь пора разъединиться и перейти к игре.

Net Tank: танковый имитатор для двух игроков

Net-Tank — игра для одного или двух игроков. Я написал ее, чтобы показать как создаются игры для нескольких участников и осветить некоторые проблемы которые могут при этом возникнуть. Это самая примитивная игра (для её написания мне понадобилось всего три дня), имеющая один-единственный уровень. Более того, это двухмерная игра, в которой игрок смотрит на поле боя сверху вниз. Мы напишем полноценную трехмерную игру типа Wolfenstetn в девятнадцатой главе, а сейчас я хочу оставить программу достаточно простой чтобы акцентировать внимание не на алгоритмах игровой логики, а на коммуникационной части.

Перед тем как мы начнем разбирать игру, я предлагаю вам сыграть в нее с кем-нибудь или, по крайней мере, в одиночку, чтобы вы понимали, о чем сейчас пойдет речь. Как я упоминал ранее, все профессионалы в области разработки игр стараются писать уникальные программы и не повторять уже существующие игры. Я модифицировал некоторые модули наших старых программ и создал на их основе новые модули. Чтобы избавиться от массивных включений исходных файлов, не имеющих непосредственного отношения к игре, я разработал несколько библиотек:

SNDLIB.C библиотека звуковых эффектов;
GRAPH1.C низкоуровневый двухмерный спрайт и графическая библиотека;
KEYLIB.C библиотека ввода с клавиатуры;
SERLIB.C коммуникационная библиотека.

Все файлы заголовков для этих библиотек имеют те же самые имена:

SNDLIB.H GRAPH1.H KEYLIB.H SERLIB.H

Все эти модули объединяются в одну обширную библиотеку с помощью менеджера LIB.EXE. Для создания этой библиотеки вам надо:

1. Скомпилировать каждый Си-модуль с помощью файла CO.BAT:

```
cl -AM  -Zi  -c  -Fc -Gs -G2 %1.c
```

2. После компиляции создается одна большая библиотека с именем MYLIB.LIB. Чтобы сделать это, вызовите библиотечный менеджер путем ввода команды

```
lib mylib.lib
```

3. Программа скажет, что библиотека с таким именем отсутствует, и спросит вашего согласия на ее создание. Ответьте утвердительно;

4. Дрбавьте в командную строку все модули, входящие в библиотеку, как это укаэано:

```
operations: +SNDLIB +GRAPH1 +KEYLIB +SERLIB
```

5. Затем появятся еще два запроса. Ответьте на них нажатием клавиши Enter.

После этого у вас появится библиотека MYLIB.LIB, которую вы можете присоединять так же, как и любую другую библиотеку.

Для создания игры, вам необходимо создать два исполняемых модуля:

- Один для игрока 1;
- Другой для игрока 2.

Чтобы это сделать, вам надо:

1. Откомпилировать NET1.C и NET2.C (2 версии игры) следующим с помощью командного файла CC.BAT:

```
cl -AM -Zi -c -Fc -Gs -G2 %1.C
if errorlevel 1 goto c fail
link  /ST:16384 /CO %1/,,,graphics.lib+myiib.iib,,
:c_fail
```

Этот командный файл компилирует игру и объединяет ее с библиотекой, чтобы создать исполняемый модуль. Сделайте это, чтобы создать два файла:

NET1.EXE и NET2.EXE;

2. Поместите NET1.EXE ,на ПК1 и NET2.EXE на ПК2. Соедините обе машины нуль-модемным кабелем через COM1;

3. Теперь можно начинать игру. Наберите NET1.EXE на ПК1 и NET2.EXE на ПК2. Машины соединятся и вы можете сразиться со своим приятелем на танковой дуэли.

Используйте следующие управляющие клавиши:

Стрелка вправо	Повернуть направо
Стрелка влево	Повернуть налев
Стрелка вверх	Двигаться вперед
Стрелка вниз	Двигаться назад
Esc	Выход из игры
Пробел	Стрельба
T	Подразнить партнера

Поиграйте в Net-Tank и попутно обратите внимание на следующие вещи:

- Возникает ли эффект запаздывания при перемещениях?
- Теряет ли игра синхронизацию?
- Если да, то в какие моменты?

Также обратите внимание на звуковые эффекты. Для их создания я использовал собственный голос и условно-бесплатную программу Blaster Master Наконец, игру можно проводить при установленной программе-ускорителе клавиатуры. Если при нажатии клавиши танк слишком резво

устремляется вперед, попробуйте уменьшить скорость реагирования клавиатуры, установив программу TURBOKEY.COM, которая имеется на дискете.

Анализ игры Net-Tank

Если вы обзовете Net-Tank пережитком каменного века, я полностью соглашусь с вами. Однако она содержит несколько интересных:технических; приемов, которые вы можете использовать (и которые в дальнейшем будут применены в Warlock'e). Вся игровая логика содержится в функции main() Си-программы, Я сделал это для того, чтобы легче было обозреть игру в целом. Исключение составляют только вызываемые функции, которые являются низкоуровневыми, но обычно их имена говорят о том, для чего они предназначены или что они делают (например, сложно не понять, что означает Draw_Sprite). Основная часть включает в себя пару сотен строк, и если вы поймете их смысл, вы в хорошей форме. Давайте разберем игру, рассматривая раздел за разделом.

Раздел 1: Инициализация

В этой части программы мы загружаем все файлы со звуковыми эффектами И графикой для игры. Элементы изображения для танков берутся из загруженный файлов и размещаются в предварительно выделенной под буфер области памяти. Net-Tank использует технику дублирующей буферизации для исключения мерцания изображения. Напомню, это означает, что изображение вначале формируется в оперативной памяти и в уже полностью подготовленном виде копируется в видеопамять. Кроме того, во время инициализации структуры данных, описывающие все игровые объекты, обнуляются или устанавливаются в исходное положение.

Уже при инициализации возникает первый вопрос сетевой игры. Программы работающие на обоих машинах почти идентичны, но должна быть небольшая разница: ваш компьютер должен показать объект противника в том же месте, где он изначально расположен на другом ПК и наоборот. Это означает, что иные позиции игровых объектов должны быть жестко установлены. Этого можно добиться, задав позиции в качестве констант в тексте программы, загружая их из неизменяемого файла данных или задавая при старте программы определенному алгоритму. Способ, с помощью которого это делается, не важен. Однако при старте программы игрок, который с точки зрения одной машины является местным, будет удаленным с точки зрения другого компыотера наоборот. Это может привести к путанице с координатами и проблемам с синхронизацией.

Раздел 2: Игровой цикл

Следующая часть начинается с главного цикла. Обратите внимание, что в игре имеется два цикла: внешний и внутренний. Внешний игровой цикл используется для инициализации некоторых переменных. Затем начинается внутренний цикл. Именно в нем и происходит основное действие.

Заметьте, что игра различает, когда она находится в состоянии соединения, а когда - нет.

Раздел 3: Удаление объектов

Как вы узнали из предыдущих глав, посвященных графике, прежде чем рисовать спрайты на новом месте, мы должны удалить их из прежней позиции. Эта часть программы убирает с игрового поля все движущиеся объекты путем восстановления ранее сохраненного фона под ними.

Раздел 4: Получение входных данных и передача состояния дистанционно управляемой системе

Здесь начинается самое приятное. Эта часть программы подразделена на два фрагмента:

- Первый из них принимает входные данные от локального игрока;
- Другой принимает входные данные от удаленного игрока.

Любопытно то, что оба фрагмента делают практически одно и то же. Разница только в том, что второй фрагмент обращает больше внимания на то, что поступает из последовательного порта, а первый в основном интересуется клавиатурой. Давайте остановимся и чуть-чуть поговорим о том, как происходит соединение. Как я говорил несколькими страницами раньше, для осуществления соединения применяются два основных метода:

- Можно передавать состояние игры в целом;
- Вы можете посылать статус устройств ввода и трактовать это как прием данных от другого джойстика или клавиатуры.

В Net-Tank я применил второй метод. Один раз в течение цикла второй машине передаются все

манипуляции игрока с клавиатурой. В это же время другой компьютер интерпретирует полученные по сети данные как действия со своей собственной клавиатурой.

Помните, чтобы этот технический прием работал, обе игры должны быть полностью детерминированы. Никаких случайностей быть не должно. В Net-Tank я полностью следовал этому правилу всюду, кроме фрагмента, изображающего взрыв. Обычно этого не сложно избежать, но иногда, после уничтожения одного из танков, игры теряют синхронизацию.

Раздел 5: Перемещение объектов

Следующая часть программы занимается перемещением объектов. Мы просто используем информацию, поступающую от локального и дистанционного ввода для передвижения и разворота танков. Интересен способ перемещения танков. Они могут двигаться в 16 различных направлениях, угол между которыми составляет 22.5 градуса. Вообще-то, чтобы переместить танк в выбранном направлении нам потребовалось бы, прежде всего, найти угол, а затем вычислить его синус и косинус для нахождения параметров передвижения.

Однако в нашей игре синусы и косинусы вычислены заранее. Это позволяет присвоить каждому из направлений свой номер (который будет совпадать с номером текущего кадра, изображающего танк) и использовать его как индекс в таблице, содержащей значения соответствующих передвижений по осям координат.

Поверьте, это лучше, чем использование математики с плавающей запятой, да еще вместе с тригонометрическими функциями. Вы узнаете гораздо больше о подобных фокусах в восемнадцатой главе, "Техника оптимизации".

Во всяком случае, как только танки двинутся, сразу же можно открывать огонь.

Раздел 6: Распознавание столкновений

Когда для всех объектов определено их новое местоположение, нужно посмотреть, не пересекаются ли они в каких-нибудь точках, то есть не нарушили ли они принцип исключительности Паули, полагающий, что две различные частицы, не могут в одно и то же время находиться в одном и том же месте. Применительно к нашей программе, мы должны ответить на вопрос: попал ли снаряд в вражеский или в наш собственный танк? Для этого координаты всех снаряда проверяются на совпадение с координатами танка. Если происходит попадание оно фиксируется с тем, чтобы в конце цикла игры изобразить взрыв.

Что касается столкновений танков друг с другом, то пока позволим им это делать беспрепятственно (предлагаю вам доработать программу в этой части самостоятельно). Кроме этого нам нужно позаботиться о том, чтобы танки не могли проходить сквозь стены. Если вы помните, игровое поле представляет собой матрицу элементов, имеющую определенную размерность. В Net-Tank размер игрового поля составляет 20x11 ячеек, каждая из которых имеет площадь 16x16 пикселей. Следовательно, чтобы увидеть, не столкнулся ли танк со стенкой, то есть попал в занятую ячейку, необходимо:

- Разделить обе координаты танка на 16;
- Округлить результат до целого;
- Использовать полученное значение как индекс ячейки игрового поля, чтобы увидеть, есть ли там блок. Если столкновение произошло, вернуть танк в прежнюю позицию.

Раздел 7: Рисование объектов

Теперь мы готовы нарисовать все объекты. Чтобы сделать это, мы должны сохранить фон в тех местах, где мы планируем разместить объекты. После этого их можно нарисовать. На этом этапе игровой экран полностью построен в дублирующем бу4эере и теперь можно посмотреть, что же у нас получилось.

Раздел 8: Дублирующий буфер

В Net-Tank используется техника дублирующей буферизации для исключения мерцания изображения. В этой части игры крайне быстрый цикл ассемблера копирует дублирующий буфер на экран. Однако мы перемещаем только первые 176 строк из буфера, .так как нижняя часть экрана неизменна и ее не надо перерисовывать.

Раздел 9: Всякая всячина

Игра прошла очередной цикл. Новая игровая ситуация отображена на экране и мы готовы вернуться к началу цикла игры. Но перед этим нам нужно сделать кое-что еще. Во время этой фазы

игры перерисовывается игровое пространство, сверкают огни взрывов, а переменные, нуждающиеся в сбросе, принимают исходные значения.

Раздел 10: Опять и снова к опять...

Условный переход к секции 1.

Итог

Мы не только научились работать последовательным портом, но и создали полную библиотеку, которая может быть использована в будущем с небольшими модификациями.

Мы также научились соединять в сеть два компьютера и узнали о факторах которые надо при этом учитывать. Наконец, у нас появилась полная (вернее почти полная) сетевая игра.

ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА

Написание видеоигр похоже на строительство дома и вам необходимо иметь подходящий инструмент для работы. В этой главе мы «заколачиваем» пиксели в графическом редакторе, «закручиваем» изображения в анимационных пакетах, делаем «электропроводку» в редакторе звуков и все это стоит на «фундаменте» нашего собственного редактора игрового пространства для игры Warlock. Ниже мы обсудим следующие моменты:

- Определение понятия инструментального средства;
- Какие инструменты нам действительно необходимы;
- Редакторы битового массива изображения;
- Пакеты для оживления изображения;
- Производство кинофильма;
- Звуковые редакторы;
- Редактор карты WarEdit;
- Использование WarEdit;
- Функциональное описание WarEdit;
- Усовершенствование WarEdit.

Определение понятия инструментального средства

Инструментом можно назвать все, что облегчает нам жизнь. Когда мы пишем игровые программы, то попутно приходится решать ряд проблем. Чтобы не повторять пройденного, нужно иметь универсальные средства для осуществления задуманного. К примеру, если нам нужно изменить силу звучания некоторого акустического эффекта, мы могли бы написать кусок программы выполняющей необходимые математические операции. Однако вряд ли мы получим удовольствие от переписывания уже созданного однажды кода, чтобы изменить громкость другого звукового эффекта. Гораздо приятнее воспользоваться программой, которая сделает все необходимое автоматически. Благо существует множество прикладных программ, облегчающих выполнение наиболее рутинной работы.

Будучи разработчиками видеоигр, мы должны подумать о том, что нам потребуется для создания игры. Далее мы должны решить, насколько трудно будет при необходимости улучшить нашу игру: видоизменить ее или добавить несколько новых уровней. После этого мысленного эксперимента можно составить список инструментов, необходимых нам для осуществления задуманного и придания игре современного вида. Наш список мог бы выглядеть примерно так:

- Редактор битовых изображений;
- Некоторое программное обеспечение для оживления битовых образов;
- Звуковой АЦП с множеством операций;
- Программа, для рисования уровней игры;и размещения объектов в игровом пространстве.

Первые три инструмента необходимы вообще любому разработчику видеоигр и я предлагаю вам по возможности расширить этот список. Если вы попробуете написать свое собственное обеспечение, вам понадобится, по крайней мере, несколько месяцев, если не пара лет!

Теперь насчет последнего в нашем списке средства. Многие игры сегодня имеют десятки, а то и сотни уровней. Составлять их вручную — дело почти безнадежное, поэтому обычно программисты пользуются специальным инструментом, создающим некую разновидность базы данных, которая и представляет собой определенный уровень. А во время игры полученная база данных просто загружается с диска. Когда один уровень пройден, точно так же загружается следующий.

Посмотрите внимательно на игру Net-Tank (мы написали ее в предыдущей главе). Вы можете заметить, что игровое поле (поле боя) в ней является двухмерной матрицей элементов, которые транслируются в обычный битовых массив, заключающий в себе саму игру. Игровое поле в Net-Tank настолько просто, что текстового редактора вполне достаточно для рисования уровня. Однако было бы очень мило иметь более совершенный инструмент, с помощью которого можно было бы рисовать поле боя и при этом сразу видеть результат. В случае с Net-Tank, на создание уровня ушло около десяти минут. Однако в играх подобных Wolfeflstein, DOOM или в нашей собственной игре Warlock давать уровни вручную немыслимо, поэтому здесь уже не обойтись без иструмента редактирования уровней.

Нас, как программистов, интересует создание специального инструмента прежде всего для тех игр, которые пишем мы сами. Этот инструмент может применяться как редактор изображений в самой программе. Наличие встроенного инструмента для создания новых уровней и нового пространства позволяет неискушенным пользователям получать все новые и новые варианты полюбившейся игры не имея ни малейшего представления о том, как она работает.

Теперь, когда мы поговорили о том, что представляют собой инструменты, давайте разберемся с возможностями того программного обеспечения, которое желательно приобрести.

Редакторы изображений

Графика и образы сегодняшних видеоигр должны быть максимально выразительными. Они должны быть объемными и выглядеть реалистично. Конечно, нет такого инструмента, который возмещает недостаток художественных способностей, а без настоящего художника ваша графика может выглядеть похожей на рисунки импрессионистов!

Можете использовать любую программу, какую пожелаете, но здесь приведен список тех минимальных средств, которыми она должна обладать:

- Желательно, чтобы программа могла работать в любом графическом режиме, и обязательно — в режиме 13h (то есть 320x200x256);
- Желательно, чтобы программа понимала различные форматы файлов как при чтении, так и при записи;
- Интерфейс должен быть удобным, позволяющим рисовать образы в увеличенном масштабе (ZOOM), если в этом возникнет необходимость;
- Цвет является важной деталью. Вам нужно иметь полный набор операций, управляющих цветовой палитрой;
- В программе должна иметься возможность вывода изображений на принтер. Неплохо, если программа будет поддерживать цветные лазерные принтеры;
- Редактор должен уметь выполнять различные геометрические преобразования изображений, такие как вращение, масштабирование, вырезание и растяжка;
- Желательно, чтобы программа поддерживала возможность борьбы с неровностями контуров (сглаживание краев) и имела бы как можно больше цветовых эффектов;

Наконец, желательно, чтобы программа имела составные страницы, и чтобы, художник мог их отрезать и склеивать, а в дальнейшем получать окончательное изображение, оперируя изготовленными ранее графическими шаблонами.

Сегодня наблюдается тенденция постепенного вытеснения битовых графических редакторов иллюстративными, которые в основном оперируют объектами и векторным представлением образов. Независимо от ваших пристрастий, поиск удовлетворительного инструментария может отнять некоторое время, прежде чем вы найдете хорошую надежную программу. Я могу посоветовать вам приобрести такие продукты как Electronic Art's Deluxe Paint или Deluxe Animation, но, в крайнем случае, подойдет PC Paintbrush.

Пакеты анимации

Этот инструмент является не таким важным, как графический редактор, однако необходимо, чтобы художественная работа выполнялась параллельно с игровым дизайном. В конечном счете, анимацию создает игровая программа, используя элементы изображений, нарисованных художником, но было бы здорово заранее увидеть, как выглядят ожившие картинки.

Вы можете решить не покупать анимационный пакет, а написать его собственноручно. Все, что вам действительно необходимо, это выбрать последовательность элементов из всех ваших битовых массивов и далее связать их в цикл с неким параметром синхронизации. Написание приличной программы могло бы отнять несколько дней, чтобы она делала это. Если вы не получаете удовольствия от ее написания, воспользуйтесь доступными пакетами. Однако цены у них - о-го-го! Это просто убийственно для наших целей.

До сих пор существует одна компания - Electronic Art, которая создает высококачественную продукцию по низким ценам. Продукт разработан исключительно для режима 13h, может быть именно поэтому он такой недорогой! Он прост и имеет инструмент для выбора по сотням опубликованных игр. Он является невероятной системой рисования в режиме 13h, и поэтому я использую большей частью именно его.

Производство кинофильма

Новой тенденцией в производстве видеоигр является использование оцифрованных образов реальных актеров и декораций. То, что я чувствую, глядя на такие игры, можно выразить так: «Ни за что!» Это же видеоигра! В ней только предполагается, что персонаж выглядит так же, как и в кино! Вообще- то оцифрованные декорации не так уж и плохи, если программа делает похожими на обычные "компьютерные" картинки и они не конфликтуют с игровыми объектами. Интересным техническим приемом, который может даже наихудшему из артистов придать великолепную внешность, является использование моделей игровых созданий совместно с камерой и устройством ввода и регистрации кадров изображения (фреймграббером). Рисунок 15.1 показывает, как можно было бы использовать реальные модели и видеоаппаратуру для этих целей.

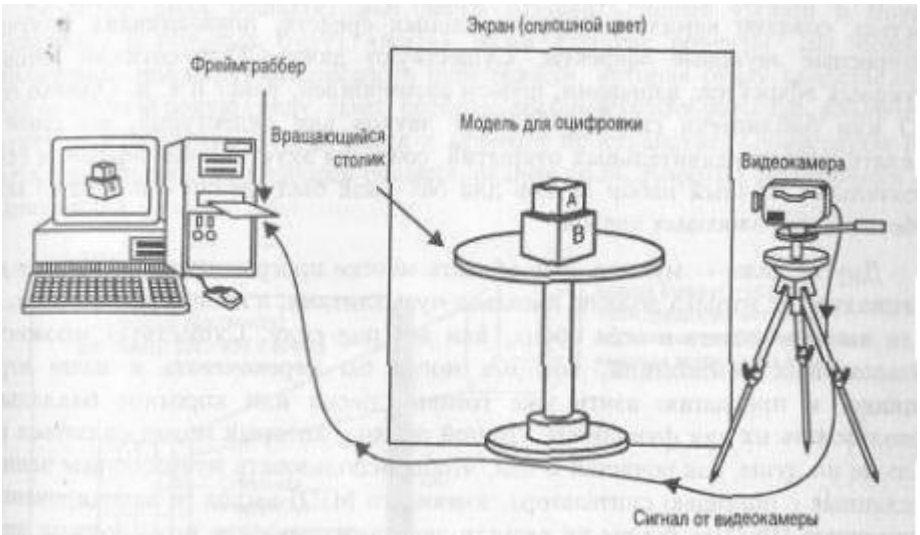


Рис. 15.1. Использование реальных моделей и видеоаппаратуры для создания художественной работы и образности.

В восьмой главе, «Высокоскоростные трехмерные спрайты», мы углубились в детали построения небольшой студии и использования видео для этого рода работы. То же самое вкратце:

- Модель некоего создания помещена на платформу с голубым, черным или белым фоном, который может быть удален позднее;
- Далее вы оцифровываете изображение, используя ПК, видеокамеру и фреймграббер.

В частности, так создавались изображения DOOM. Модельер изготавливает модели созданий, потом помещает их на платформу и оцифровывает. Затем фиксирует их цвета и отдает программистам, чтобы те использовали их в игре. Следовательно, получение различных ракурсов трехмерных объектов является таким же легким делом, что и поворот или передвижение модели. Потому если у вас есть немного денег, и ваши художественные таланты не зачахли стоит попробовать пойти этим путем.

Звуковые редакторы

Существуют две разновидности звуков в видеоигре:

- Синтезированные эффекты;
- Музыка.

Поскольку синтезированные эффекты пользуются большой популярностью, создано немало инструментальных средств, позволяющих получать интересные звуковые эффекты. Существуют даже CD с сотнями готовых звуковых эффектов: взрывами, шумом автомобилей, ракет и т. д. Однако если CD или библиотека синтезированных звуков вам недоступна, вы сможете сделать немало удивительных открытий, создавая акустические эффекты самостоятельно. Полный набор звуков для Net-Tank был сделан с помощью моих собственных голосовых связок!

Другое дело - музыка. Эту область многие программисты оставляют для специалистов: музыка должна писаться музыкантами, а не программистами. Но если вы совмещаете в себе обоих, вам все под силу. Существует множество музыкальных композиций, которые могли бы перекочевать в ваши игры. Однако, я предлагаю взять уже готовые песни или короткие баллады и использовать их как фундамент. Другой подход, который может оказаться вам больше по душе, заключается в том, чтобы использовать музыкальные записи, сделанные с помощью синтезатора, имеющего midi-выход, и закодированные с помощью ПК. Что бы вы ни решили делать, музыка для игры должна иметь собственный темп, а «характер» музыки должен соответствовать каждому уровню и набору обстоятельств. По этим причинам многие программисты заключают контракты с профессиональными музыкантами по поводу написания музыкальной части игры, чтобы данная работа была выполнена с наилучшим качеством. Следовательно, вам может понадобиться приобрести или создать самим достаточно «продвинутый» звуковой драйвер. Однако если вы считаете, что вам не нужны сложные звуковые эффекты, то дигитайзер, несколько музыкальных примеров и определенный творческий настрой дадут нужные результаты.

Редактор поля WarEdit

Мы говорили о некоторых доступных инструментальных средствах, которые необходимы программистам игр и о том, как они используются. Теперь я хочу рассказать о тех инструментах, которые помогут в создании нашей собственной игры. Мы будем делать трехмерную игру под названием Warlock. Она будет очень проста, и я до сих пор даже не уверен в том, что в ней будут хоть какие-нибудь противники. Тем не менее, я собираюсь изготовить инструмент который позволит нам легко создавать уровни для игры.

Игровое пространство Warlock представляет собой трехмерный мир, с набором кубов, вертикальные грани которых разукрашены определенной фактурой. Общая площадь нашего мира

приблизительно составляет 200x200 сторон кубов как показано па рисунке 15.2. Мы могли бы представить это пространство виде текстового файла и набить 40000 символов. Однако я не думаю, что сам , был бы этому рад. Вместо этого я решил создать простой редактор поля, который позволяет рисовать план игрового пространства, рассматриваемого сверху.

Редактор поля позволит нам легко создавать новые уровни и новые обстановки», в которых будут «жить» наши игровые объекты. Вы можете использовать мышь, чтобы рисовать поверхность, которая будет представлять собой полную игровую среду. Цвет, которым вы рисуете изображение, различает структуры и объекты, находящиеся в игровом пространстве. Существуют несколько управляющих функций редактирования поля. Коротко остановимся на их операциях.

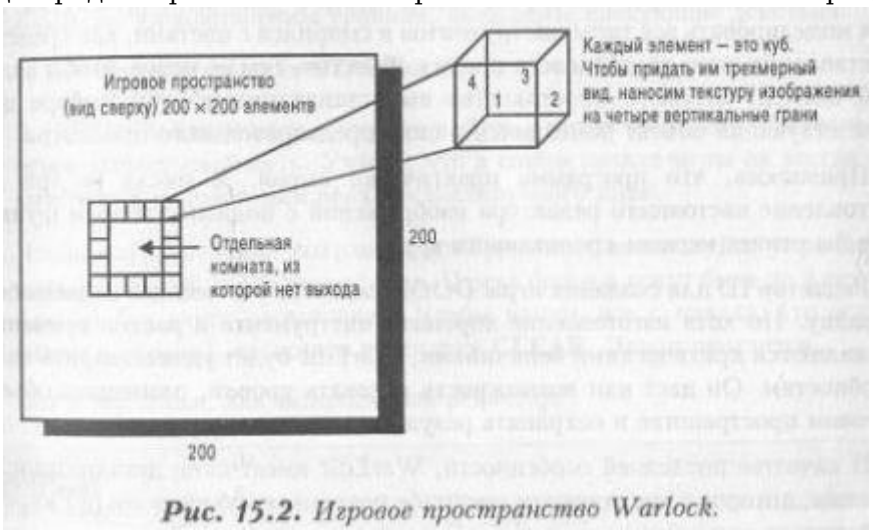


***Рис. 15.2.** Игровое пространство Warlock.*

Я решил представить игровое пространство, используя технику базовых элементов. Следовательно, все что мне нужно сделать, это создать двухмерную матрицу, представляющую собой игровое пространство, и заполнить ее целыми бедами, которые будут обозначать типы объектов, находящихся в соответствующих позициях. Объектами могут быть стены, «пища», «лекарство», монстры, «свитки» или двери.

Теперь, если стена или дверь помещены в определенную ячейку матрицы, они заполняют квадрат полностью. Однако есть и другие объекты, такие, наример, как «пища» или «свиток», которые имеют гораздо меньшие размеры, чем стена (которая в действительности представляет собой шестигранный куб; верх и низ его никогда не видны). Помещая один из таких объектов на карту игрового поля, мы можем разместить его только где-то внутри квадрата. Более точно определить его положение мы не в состоянии. Когда программа встречает в базе данных такой объект, она случайным образом помещает его внутрь квадрата. Лучшим решением было бы создать редактор, который имеет более передовую структуру данных, с тем, чтобы мелкие объекты могли быть размещены более аккуратно. Однако того, что редактор делает сейчас, для наших целей вполне достаточно.

Вы можете создавать уровни, рисуя стены, двери и т. п. различными цветами, выбирая объект путем изменения цвета. Я применил цвета для пред. ставления объектов потому, что это самый легкий путь моделирования, хотя использование уменьшенных образов (иконок) объектов был бы, конечно намного предпочтительнее. Но при таком подходе сразу же возникает проблема которая заключается в том, что для представления матрицы размером 200x200 иконок понадобилось бы поле, во много раз превосходящее размеры экрана и для работы с ним потребовалось бы позаботиться о механизме прокрутки. Я не хотел моделировать все типы инструментов и смирился с цветами, как средством представления стен, дверей и всех прочих объектов. Тем не менее, чтобы видеть какой элемент игрового пространства вы устанавливаете, при выборе цвета соответствующий объект помещается в окно предварительного просмотра.

Признаюсь, что программа практически сырая. Я писал ее три дня. Изготовление настоящего редактора изображений с полным набором функций могло бы отнять месяцы кропотливого труда.

Редактор ID для создания игры DOOM отнял шесть месяцев на разработку и отладку. Но хотя изготовление хорошего инструмента и расход времени на него являются критическими величинами, WarEdit будет удовлетворять нашим потребностям. Он даст нам возможность рисовать уровни, размещать объекты в игровом пространстве и сохранять результат на диске.

В качестве последней особенности, WarEdit имеет окно детализации изображения, которое в увеличенном масштабе показывает область вокруг курсора. Это помогает размещению дверей и созданию тонких деталей.

Использование WarEdit

Для запуска WarEdit введите wedit в строке DOS. Эта программа начинает работу с загрузки текстур изображений и всех образов в окно предварительного просмотра. Как она это делает, вы увидите по вспышкам изображений на экране. Как только загрузка текстур закончится, на экране появится интерфейс пользователя.

Управление WarEdit несложно. Чтобы начать рисовать изображаемы уровень, необходимо

выполнить следующие действия:

1. Укажите курсором мыши на выбранный объект и щелкните левой кнопкой. Объекты расположены в правой части экрана. Это стены, двери и т. д. Чтобы увидеть предварительный вид текстуры изображения или объекта любого цвета, попробуйте переключить этот цвет. Я старался подбирать цвета так, чтобы они были по возможности ближе к цветам изображений и текстур, которые он представляет (к примеру, серые цвета для «каменных» текстур). Но оттенки, естественно, уже не относятся к делу. Более темный оттенок не означает более темную текстуру изображения. Он означает просто другую текстуру.
2. Выбрав цвет и, в конечном счете, текстуру изображения объекта, переместите курсор мыши в большое окно с левой стороны экрана. Эта область предназначена для рисования уровня.

Я предлагаю прежде всего нарисовать все стены и двери. Когда уровень, готов, разместите ваши собственные игровые объекты. Если вы сделаете ошибку, то используйте правую кнопку мыши как стирательную резинку.

Когда вы удовлетворены уровнем, выполните следующие действия:

1. Щелкните по кнопке START в нижнем правом углу.
2. Поставьте одиночную точку где-нибудь в игровой области. Из этого места игрок начнет свой путь. Учтите, что в самом начале игры он всегда будет смотреть в направлении верха редактируемого поля.

Чтобы загрузить или сохранить изображение, используйте управление в верхней правой части интерфейса. Имена файлов могут быть до 8 символов длиной без учета расширения. Чтобы начать все с начала, (то есть для очистки экрана), щелкните по кнопке CLEAR. Экран очистится.

Вот и все о том, как использовать редактор.

Примечание

У вас может быть более поздняя версия WarEdit'a. Однако база данных, которую создает программа, описанная в этой книге, будет почти полностью совместима с любой более новой версией WarEdit'a. Я гарантирую, что оформление и текстуры будут другими, цвета тоже могут измениться. Поэтому, если вы обнаружите, что ваши объекты в новой версии редактора выглядят как-то иначе, не обращайте на это внимания — новый WarEdit будет способен загрузить ваши уровни. Я мог бы Дать вам последний вариант программы прямо сейчас, однако, прилагаемая версия более надежна в работе. В следующей главе вы увидите, какие изменения в редактор нужно ввести и зачем. Если я их реализую, то вы о них обязательно прочтете в аннотации к программе.

Теперь, когда мы поговорили о WarEdit с точки зрения пользователя, давайте обсудим его структуру и формат базы данных.

Функциональное описание программы WarEdit

Программа WarEdit исключительно проста для понимания. В основном, бопьшинство инструментов не слишком сложно создать и их описание только отнимает время. Единственная убийственно сложная вещь — это графический интерфейс Поскольку мы используем DOS, а не Windows, то должны создать все сами. Например, если мы хотим увидеть в программе кнопки, придется написать функции для их изображения и обслуживания. То же самое относится и ко всем прочим элементам интерфейса. Тут уж ничего не поделаешь, с этим нужно считаться.

Если вы собираетесь изготавливать инструменты с графическим интерфейсом, лучше начать с создания библиотеки, которой вы сможете доверять. Для этого вам придется написать тысячи строк программы.

Попробуем подойти к процессу дизайна с другой стороны. Посмотрим, что необходимо для воплощения конкретно WarEdit'a. Затем мы охватим несколько наиболее интересных функций.

WarEdit был разработан для создания игрового пространства Warlock, основываясь на тех текстурах и объектах, которые могут присутствовать в данной игре. Нам нужно иметь возможность загружать и сохранять уровни, а также просматривать предварительное изображение текстур. Я сознаю, что для большинства людей отдельные пиксели для представления блоков покажутся слишком мелкими, так как цвет одного пикселя трудно будет отличить от цвета другого, стоящего рядом. Приняв это во внимание, я решил создать окно детализации изображения, которое должно помочь различать близлежащие объекты. Нам понадобятся следующие функции:

- Для загрузки PCX-файлов;
- Для рисования спрайтов и масштабирования объектов;
- Библиотека функций мыши;
- Диалоговые окна;
- Процедура определяющая положение курсора мыши в момент щелчка;
- Функция для детализации изображения;
- Некоторые функции ввода/вывода для загрузки и-сохранения данных;
- Интерфейс пользователя.

Мы уже написали первые три функции. Остальные я собираюсь создавать от нуля или делая добавления к уже имеющейся программе.

Давайте начнем с последнего. Я догадывался, что на разработку алгоритма графического

интерфейса у меня уйдет, по меньшей мере, несколько дней. Поэтому я решил просто нарисовать его в редакторе Deluxe Paint и загрузят в виде PCX-файла. Закончив изготовление интерфейса, я записал местоположение размеры всех окон и образов, чтобы впоследствии задать все интересующие области в виде констант. Самым сложным моментом был выбор цветов. Я решил пользовать в игре четыре различных типа стенок, причем так, чтобы каждый этих типов мог быть представлен шестью видами текстур. Поэтому для рисования стен можно выбрать четыре различных цвета: серый, красный, синий, зеленый, а для представления текстур подобрать оттенки этих основных цветов.

Зная, что пользователю может показаться трудным различать на экране каждый из оттенков, я решил создать окно детализации изображения. Небольшая площадь редактируемого игрового пространства вокруг курсора мыши захватывается и в увеличенном масштабе помещается в окно детализации. При этом каждый пиксель изображается квадратиком 3x3 пикселя.

При выборе цвета, представляющего собой определенную текстуру или объект, вы видите изображение выбранного объекта в окне предварительного просмотра. Это выполняется посредством взятия битовой карты текстуры или объекта и изменения ее масштаба до необходимого размера. Теперь цветом можно рисовать изображения. Однако когда вы рисуете объекты игрового пространства, в базу данных передается не само значение цвета. С использованием справочной таблицы цвет переводится в значение, распознаваемое программным кодом игры, как обычные данные текстуры или объекта. Листинг 15.1 содержит определения типов этих данных.

Листинг 15.1. Секция определений WarEdit.

```
#define WALLS START        64
#define NUM_WALLS         24
#define DOORS START       128
#define NUM_DOORS          4
#define SCROLLS_START      144
#define NUM_SCROLLS        4
#define POTIONS_START      160
#define NUM_POTIONS        4
#define FOODS_START        176
#define NUM_FOODS          2
#define MONSTERS_START     192
#define NUM_MONSTERS       2
#define WALL_STONE_1      (WALLS_START+0)   // Пока только 6
#define WALL_STONE_2      (WALLS_START+1)
#define WALL_STONE_3      (WALLS_START+2)
#define WALL_STONE_4      (WALLS_START+3)
#define WALL_STONE_5      (WALLS_START+4)
#define WALL_STONE_6.      (WALLS_START+5)
#define NUM_STONE_WALLS   6
#define WALL_MELT_1     (WALLS_START+6) // Пока только 6
#define WALL_MELT_2     (WALL3_START+7)
#define WALL_MELT_3     (WALLS_START+8)
#define WALL_MELT_4     (WALLS_START+9)
#define WALL_MELT__5    (WALLS_START+10)
#define WALL_MELT_6     (WALLS__START+11)
#define NUM_MELT_WALLS 6
#define WALL_OOZ_1      (WALLS_START+12) // Пока только 6
#define WALL_OOZ_2      (WALLS_START+13)
#define WALL_OOZ_3      (WALLS_START+14)
#define WALL_OOZ_4      (WALLS_START+15)
#define WALL_OOZ_5      (WALLS_START+16)
#define WALL_OOZ_6      (WALLS_START+17)
#define NUM_OOZ_WALLS   6
#define WALL_ICE_1      (WALLS_START+18) // Пока только 6
#define WALL_ICE_2      (WALLS_START+19)
#define WALL_ICE_3      (WALLS_START+20)
#define WALL_ICE_4      (WALLS_START+21)
#define WALL_ICE_5      (WALLS_START+22)
#define WALL_ICE_6      (WALLS_START+23)
#define NUM_ICE_WALLS   6
#define DOORS_1         (DOORS_START+0) // Пока только 4
#define DOORS_2         (DOORS_START+1)
#define DOORS_3         (DOORS_START+2)
#define DOORS_4         (DOORS_START+3)
```

```
#define SCROLLS_1       (SCROLLS_START+0) // Пока только 4
#define SCROLLS_2       (SCROLLS_START+1)
#define SCROLLS_3       (SCROLLS_START+2)
#define SCROLLS_4       (SCROLLS_START+3)
#define POTIONS_1       (POTIONS_START+0) // Пока только 4
#define POTIONS_2       (POTIONS_START+1)
#define POTIONS_3       (POTIONS_START+2)
#define POTIONS_4       (POTIONS_START+3)
#define FOODS_1         (FOODS_START+0) // Пока только 2
#define FOODS_2         (FOODS_START+1)
#define MONSTERS_1      (MONSTERS_START+0) // Пока только 2
#define MONSTERS_2      (MONSTERS_START+1)
#define GAME_START      255 // точка, из которой игрок
                        // начинает свой путь
```

Как можно видеть из Листинга 15.1, для каждой текстуры или объекта игрового пространства выделен некоторый диапазон значений. К примеру, все стены будут кодироваться числами от 64 до 127. В данный момент существует только 64 стены, но я оставил в программе место еще для 64 стен. Такой же подход применен и для определения всех остальных игровых объектов, так что структуру данных можно улучшить без особого труда. Когда вы рисуете карту игрового поля, то в действительности заполняете базу данных. Она представляет собой длинный массив, по способу доступа напоминающий двухмерную матрицу. Когда массив построен и уровень закончен, мы должны как-то сохранить данные. Для этого предусмотрены кнопки LOAD и SAVE. При нажатии одной из них, появляется диалоговое окно, запрашивающее подтверждение команды. Если вы выбрали YES, с помощью стандартной функции fopen() открывается файл, после чего данные сохраняются в файле или загружаются из файла. Но предварительно нужно ввести имя файла, используя строчный редактор. Этот редактор мне пришлось сделать самостоятельно. Он очень прост и понимает только алфавитно-цифровые клавиши.

Примечание

Мне пришлось написать такой редактор самостоятельно, поскольку единственный способ ввода строки, предлагаемый языком Си - это функция scanf (). Однако использование этой функции чревато неприятностями: при наборе возможны ошибки, текст может выйти за границы диалогового окна и т. д.

Давайте поговорим о диалоговом окне, о котором я только что упоминал. Оно представляет собой PCX-файл с двумя нарисованными кнопками. Хотя кнопки, конечно же, являются обычными картинками, я точно знаю их расположение относительно левого верхнего угла. Все, что мне нужно сделать, это доверить, находится ли мышь над кнопкой и щелкнули ли вы мышью. Для осуществления этого теста я создал универсальную функцию, которой нужно передать следующие параметры:

- Размер каждой кнопки;
- Количество кнопок в столбце и строке;
- Местоположение первой кнопки;
- Расстояние между кнопками.

Эта функция будет определять, какая из кнопок нажата. (Вспомните, не существует легкого пути для получения подобных вещей. Мы все должны сделать собственноручно. У нас нет приличного менеджера окон и механизма посылающего нам сообщения при нажатии кнопок. Хотя вы можете попробовать написать его сами, поскольку в мои планы это не входит.) В Листинге 15.2 показана функция, которая обнаруживает нажатие кнопок.

Листинг 15.2. Обнаружение нажатия кнопки.

```
int Icon_Hit(int xo, int yo, int dx, int dy,
int width, int height,
int num_columns, int num_rows,
int mx, int my)
{
// получая геометрические установки кнопок, эта функция вычисляет,
// по которой из них щелкнули мышью
int row, column, xs,ys,xe,ye;
for (row=0; row<num_rows; row++)
{
// вычислить начало и конец Y-координаты колонки
ys = row*dy + yo;
ye = ys+height;
for(column=0; column<num_columns; column++)
```

```
{
xs = column*dx + xo;
xe = xs+width;
//Проверить, находится ли курсор мыши в области кнопки
if (mx>xs && mx<xe && my>ys && my<ye)
{
return(column + row*num_columns);
} // конец if
} // конец внутреннего цикла
} // конец внешнего цикла
return(-1); // не нажата
} // конец функции
```

Это все относительно редактора WarEdit. Я предлагаю вам поиграть с программой, чтобы получить представление о ней, прежде чем мы начнем разговор об улучшениях, которые могут быть внесены в редактор.

Наконец, для создания работающей версии WarEdit желательно использовать два объектных файла, называемых GRAPH0.OBJ и MOUSELIB.OBJ. Я предлагаю объединить их в библиотеку и затем связать с WEDIT.C. (Хотя, конечно, вы можете поступать, как вам больше нравится). Как вы понимаете, файл GRAPH0.OBJ получается после трансляции исходников GRAPH0.C и GRAPH0.H, а MOUSELIB.OBJ - из MOUSELIB.C, о котором уже упоминалось ранее.

Улучшения WarEdit

WarEdit уже будет работать вполне удовлетворительно, но на полноценный редактор он все же не тянет. В нем недостает многих деталей, необходимых для настоящей работы. Сейчас я хочу сказать о некоторых моментах, которые вы могли бы доработать сами:

- Использование разноцветных точек для представления игровых объектов слишком примитивно. Более серьезным и удобным решением было бы использование окна редактирования с возможностью прокрутки, в котором каждый элемент игрового пространства представлен собственной уменьшенной копией;
- В нашем редакторе любой объект занимает всю ячейку игрового пространства целиком. Но что если мы хотим поместить два мелких предмета в одно и то же место? Здесь могло бы пригодиться окно детализации ячейки;
- А как быть насчет спецэффектов типа освещения? Неплохо было бы иметь возможность изменять уровни освещения в целом;
- Еще не помешало бы добавить возможность указания начальных и конечных точек перемещения (телепортации) игрока;
- Важный технический прием, не примененный в WarEdit, называется полиморфизмом. Эта методика позволяет, используя одну и ту же ссылку, обращаться к различным типам данных. Все структуры данных и определения обычно жестко закодированы в программе и их необходимо импортировать в игру в виде включаемого файла. Но гораздо лучше иметь некоторое подобие синтаксического анализатора, который будет считывать все игровые объекты и значения из файла инициализации.

Можно было бы и дальше продолжить список усовершенствований редактора. Обычно разработчики так и поступают, пока не получат окончательную версию. Фактически, мы могли бы потратить несколько дней и улучшить редактор до уровня Wolfenstein 3-D. (Добавление в него деталей из DOOM отняло бы месяцы). В любом случае, я думаю, у вас уже есть множество идей на этот счет, и если ваш первый редактор окажется вдвое лучше WarEdit'a, то с ним, вероятно, вы создадите и вдвое лучшую игру.

Итог

Мы уже давно путешествуем, и если вы все еще со мной, то, должно быть, относитесь к тем немногим избранным, чье предназначение - создавать иные миры для ПК. Бьюсь об заклад, что у вас пока еще нет всех знаний из громадного числа областей, нуждающихся в изучении. Видеоигры действительно трудны для написания по своей природе и любой, кто может их создавать, достиг великой цели. Надеюсь, и вы будете из их числа...

ГРАФИЧЕСКОЕ ОФОРМЛЕНИЕ ИГРЫ

Существует множество различных графических программ, как свободно распространяемых, так и продаваемых за деньги. Каждая из них имеет свои сильные и слабые стороны. Иногда графический пакет может состоять из нескольких программ для обеспечения всех функций и особенностей, необходимых для разработки вашей игровой графики.

Среди свободно распространяемых известны такие программы как MVP Paint, NeoPaint, Desktop Paint 256. Они обладают, одним достоинством, увеличивающим объем , их использования: вы можете «попробовать их, прежде чем скушать». Здесь уместно лишний раз подчеркнуть, что даже за условно бесплатные программы все-таки надо платить.

Кроме того, существует огромное количество коммерческих графических программ. Цены на них колеблются от десятков до тысяч долларов. Один из пакетов, имеющих основание высокую цену и необычайно популярных среди разработчиков игр, называется Electronic Arfc's Deluxe Paint II Enchanced. Существует также программа, создающая настоящие фильмы - Deluxe Paint Animator. Весьма популярны программы, которые комбинируют растровую и векторную графику. К ним относятся, например, Corel DRAW и Micrografx Designer. Другим изобретением и мощной графической программой является Fractal Design Painter, который замечательно имитирует традиционное изобразительное искусство. На самом верху находятся программы для создания настоящих мультфильмов типа Autodesk Animator Pro и 3D Studio предлагающие неограниченные возможности в разработке изображений и мультипликации.

Удобные детали и инструменты

На что вам в первую очередь необходимо обратить внимание при выборе графической программы? Существует не так уж и много возможностей графических редакторов, которые я использую наиболее часто. Я с удовольствием помогу вам и укажу на них:

- Посмотрите, есть ли в программе возможность импорта и экспорта различных графических форматов. Существует целое семейство битовых графических форматов, и у вас должна иметься возможность отконвертировать любое изображение в тот формат, который вы в конечном счете используете. К примеру, вы можете решить, что ваши окончательные картинки должны сохраняться в PCX-файлах, но нашли несколько интересных изображений в GIF-, BMP- или TIF-формате. Было бы прекрасно иметь возможность преобразовать их в выбранный формат, при необходимости отредактировать и сохранить в новом PCX-файле. Эти четыре формата наиболее часто употребляются графическими редакторами, поддерживающими 256-цветный режим;
- Обратите внимание на способ управления цветами и их расположение в палитре. У вас может возникнуть необходимость собрать окончательное изображение из нескольких исходных картинок, и вам потребуется создать палитру, которая будет обеспечивать наилучшие результаты с различными типами изображений. Вы можете обнаружить, что комбинирование этих изображений приводит к перемешиванию палитры. Возможность перестановки цветов по яркости оттенков является удобной деталью. Мы еще вернемся к этой возможности и рассмотрим ее чуть позже в данной главе;
- Очень хорошо, если программа позволяет уменьшить количество цветов в изображении. Вы быстро обнаружите, что 256 цветов вам хватит ненадолго, если вы работаете с несколькими фотографиями или набором текстур. Вам может потребоваться уменьшить количество цветов, используемое каждым изображением, до 64 или даже до 32;
- Очень хорошей особенностью некоторых программ является способность комбинировать палитры двух и более изображений и подбирать лучшие цвета для объединенного изображения. После сокращения палитр отдельных изображений (как это было описано в предыдущем пункте) вы сможете собрать несколько, меньших палитр в одной. Если вы не найдете такую программу, воспользуйтесь редактором, поддерживающим 24-разрядную графику (реальные цвета). Этот тип программ, позволит вам отрезать, и приклеивать различные изображения в одном файле, а затем преобразовать собранное изображение в вашу лучшую 256-цветную палитру;
- Вам также может потребоваться возможность создания плавных цветовых переходов. Существует множество таких программ и некоторые из них работают с двумя цветами, плавно изменяя окраску от одного оттенка к другому. Иные графические программы позволяют выбрать диапазон цветов в палитре и произвольно смонтировать их. Я предпочитаю последний метод, поскольку он дает наиболее хороший результат и особенно удобен для раскраски битовых образов. Одна из программ, обладающих таким достоинством, называется Electronic Art's Deluxe Paint;
- Наконец, вы будете очень довольны, если сможете выполнять сглаживание. Это такой технический прием, который позволяет убирать неровности внешнего вида скошенных или закругленных областей путем смешивания промежуточных цветов в шероховатых местах. Если вы используете эту особенность осторожно, то можете сгладить цветовой

переход в наиболее контрастных участках и придать изображению более натуральный внешний вид.

Однако существует ряд важных «инструментальных средств», которые невозможно купить ни за какие деньги. Это желание, терпение и хороший художественный вкус. Доведение до совершенства любой задачи требует практики, будь то программирование, кулинария или рисование. Не отчаивайтесь, если ваше первое творение окажется совсем не таким, каким вы его себе представляли. Считайте, что, сделав это, вы уже чему-то научились и двигайтесь вперед.

Теперь давайте рассмотрим основы создания игровой графики.

Первый этап: планирование

На первом этапе вам надо будет разработать план игры и составить список тех объектов, которые вы будете рисовать. Этот список включает в себя фактуры для изображения стен, персонажи и игровые объекты. Прежде чем начать рисовать картинки, вы должны создать палитру, содержащую все используемые в игре цвета. Иногда это довольно просто, особенно в случае, когда все фактуры и персонажи рисуются от руки. Однако, если вы намереваетесь использовать графику, приобретенную из других источников, таких как фотографии и оцифрованные кинокадры, разработка палитры потребует некоторого планирония.

До начала разработки палитры вам необходимо принять во внимание некоторые вещи относительно игры, для которой создается графика. Ниже обсуждается ряд моментов, о которых нужно всегда помнить.

Разрешающая способность и количество цветов

Некоторые игры используют 16 цветов, другие - 256. В этой книге мы будем концентрироваться на разработке 256-цветпых игр. Разрешение экрана может быть 320x200, 320x240, 640x480 и выше. Применение высокой разрешающей способности связано с рядом проблем достижения необходимого быстродействия, если только вы не используете высокопроизводительный компьютер но принципы разработки графики в любом случае остаются одними и теми же.

Для кого вы пишете игру?

Подумайте, для кого предназначена ваша игра: для детей, подростков, или взрослых? Детские игры обычно выполняются в сочных, красочных тонах. Вспомните, что детей особенно привлекают мультфильмы, цирк и парады. Во всем этом присутствуют яркие краски. Подросткам обычно нравятся игры со множеством жутких монстров — чем больше, тем лучше. В играх, предназначенных для более старшей аудитории,, наибольшее внимание, как правило уделяется содержанию игры.

Содержание и общее настроение игры

Подумайте, что будет представлять собой ваша игра. Если это будет, например, космическое путешествие, то может потребоваться комбинация ярких и темных тонов: яркие будут представлять высокотехнологичные металлические конструкции, а более темные - звездные дали космического пространства. Средневековые сцены можно передать богатой палитрой земляных красок для отображения прошлого. Игры-ужастики могли бы, вероятно, потребовать темных, мрачных тонов. Если в вашей игре должны встретиться сцены как дневного, так и ночного освещения, это также необходимо учесть при разработке палитры.

Что детализировать, а что нет?

Возникающие в игре ситуации заставляют использовать при разработке палитры определенные сочетания цветов. Если у вас есть несколько фотографий или трехмерных изображений, которые вы собираетесь использовать, желательно показать их в игре максимально приближенными к оригиналу. Позже я расскажу, как можно оптимизировать палитру, объединяющую столько различных оттенков, сколько требуется для придания изображению максимально правдоподобного вида.

Несколько слов по поводу цвета

С точки зрения художников, большинство оттенков может быть получено из трех первичных цветов: красного, желтого и синего. Если вы смешиваете эти три краски в одинаковых пропорциях, то вы получите цвет, очень близкий к черному. Единственный цвет, который нельзя получить смешением красок - это белый.

В компьютерном мире, где цвет «соткан» из света, различные оттенки получаются путем изменения уровней красного, зеленого и синего цвета. Многие программы описывают цвета на вашем экране, представляя их тремя номерами, отделяемыми запятой. Например, черный цвет может быть представлен как R0, G0, B0 (это означает, что для данного цвета все три составляющие цвета отключены). Белый цвет создается путем установки всех трех значений на максимум (R63, G63, B63 - для 8-битных и R255, G255, B255 - для 24-битных регистров).

Цвета могут быть условно разделены на «холодные» и «теплые». В таблице 16.1 показано несколько примеров.

Таблица 16.1 «Температуры» цветов.

Теплые цвета	Холодные цвета
красный	зеленый
желтый	синий
оранжевый	фиолетовый
земляные краски	все оттенки серого, включая белый и черный

В связи с этим имеется одна хитрость. Теплые цвета могут быть «охлаждены» смешиванием, с холодными и, наоборот, холодные краски можно «подогреть», добавляя к ним теплые тона. Вообще-то, это очень неплохая идея - издавать декорации, раскрашенные как в холодные, так и в теплые цвета. Это придаст изображению более приятный для глаз вид.

Стоит упомянуть еще вот о чем. Любое изображение может восприниматься совершенно по-разному в зависимости от цвета фона. Чтобы увидеть это на примере, просмотрите файл EXAMP16.PCX (который вы можете найти на дискете). Этот файл показывает двух чудовищ, одного серого, а другого красного, которые помещены на светлосером и белом фоне. В обоих случаях чудовища имеют один и тот же цвет, но, будучи помещенными на белом фоне, выглядят темнее.

Вы должны экспериментальным путем определить, на каком фоне теплые цвета выглядят похожими на холодные и наоборот. Нельзя допускать, чтобы цвета «дрались» друг с другом и утомляли ваши глаза. Главной причиной такого подбора является необходимость сохранения одного и того же цвета предмета во всех сценах.

Разработка 256-цветной палитры

Существует несколько различных подходов к разработке палитр для 256-цветных игр. Один метод, обычно используемый в играх приключенческого типа заключается в том, чтобы определить часть палитры, неизменной для всех изображений. Эти цвета используются для изображений, появляющихся более чем на одном игровом экране (например, это могут быть элементы пользовательского интерфейса, персонажи и некоторые другие объекты). Обычно для этого достаточно 64 цветов. Оставшиеся 192 цвета палитры будут изменяться от экрана к экрану в зависимости от декораций, сопровождающих игру. Другой подход состоит в создании единой палитры, используемой для всех пейзажей, объектов и персонажей игры. Этот метод мы здесь и обсудим.

Наилучшие для использования цвета

Если вы разрабатываете все изображения своей игры с нуля, у вас есть обширный выбор цветов для создания собственной игровой палитры. Однако, как правило, вы будете применять следующие диапазоны цветов:

- Оттенки серого (от чисто-белого до чисто-черного);
- Все оттенки основных цветов (красного, желтого, синего);
- Оттенки каждого из вторичных цветов (оранжевого, зеленого, фиолетового);
- Телесные тона для персонажей;
- Земляные краски для земли, леса и т. д.

Последовательный порядок цветов

Что я понимаю под словами «диапазон цветов» или «последовательный порядок цветов»? Рассмотрите файл EXAMP01.PCX. К примеру, вы хотите изобразить красный шар так, чтобы он выглядел действительно круглым, объемным. Он должен располагаться на черном фоне и освещаться с правой фронтальной стороны. На плоскости этот шар был бы представлен кругом, заполненным однородным цветом, средним оттенком красного. Такое представление шара показано в верхней части на рисунке 16.1.

Едва ли это убеждает, не так ли? Чтобы шар выглядел объемным, оттенки красного должны плавно переходить от светлого к темному, что придаст объекту соответствующую форму и подсветку. Тот же самый технический прием применяется к прямоугольным и к цилиндрическим формам, как показано на рисунке.

Вот такой плавный переход оттенков одного цвета от светлого к темному и называю диапазоном или последовательным порядком цветов. Рассматривая файл EXAMP01.PCX, обратите внимание, что палитра для него имеет несколько различных цветовых диапазонов. Попробуйте поэкспериментировать с это палитрой и изменить ее в соответствии с вашим цветовым предпочтением.

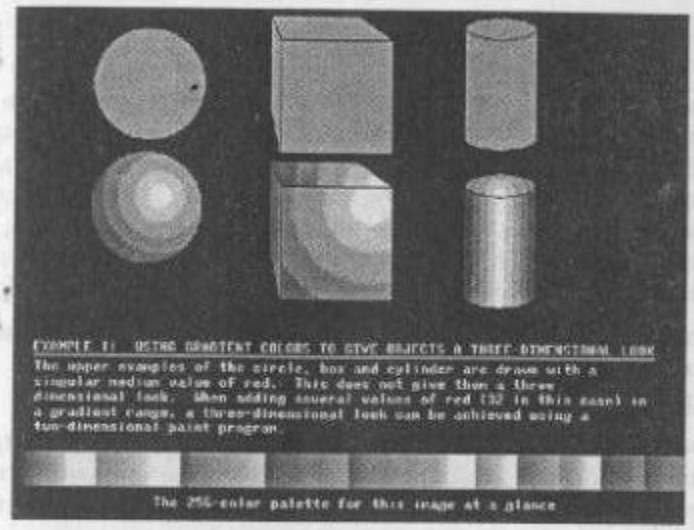


Рис. 16.1. Использование градиентных цветов для достижения объемности изображения с двухмерной программой рисования (см. файл EXAMP01.PCX на прилагаемой дискете).

Сохранение цвета для прозрачности

При разработке палитры ни в коем случае нельзя забывать о том, что вы должны зарезервировать «прозрачный» цвет для вывода спрайтов. Если увидите пустоту в каком-то изображении, то, скорее всего, вы допустили ошибку. Помните, что «прозрачный» цвет при наложении спрайтов не изменит цвет фона.

Применение палитр других изображений

Существует немало коммерческих библиотек готовых изображений и фотографии, которые можно использовать без выплаты дополнительного вознаграждения их создателям. Однако, чтобы быть уверенным в том, что у вас действительно есть право на использование этих изображений в собственных программах, необходимо прежде всего очень внимательно прочитать липензионное соглашение для таких коллекций. Если у вас возникнут на этот счет какие-либо сомнения, всегда входите в контакт с издателем библиотеки.

Ни в коем случае не рекомендуется использование изображений, взятых из ББС или других коммуникационных служб. Многие находящиеся там графические файлы были просто скопированы из журналов, передач и фильмов без соблюдения соответствующих авторских прав. Постарайтесь ничего не испольвать в вашей игре без разрешения, полученного из сопроводительной документации. Безопаснее всего, конечно, употреблять только собственноручно изготовленные изображения и фотографии, либо, если нет возможности заниматься этим самостоятельно, нанять кого-нибудь специально, для разработки графики.

В предыдущем разделе мы обсуждали, какие цвета лучше всего размещать в палитре, чтобы иметь наиболее широкие возможности использования цветового диапазона. Если вы попытаетесь какую-либо фотографию подогнать под имеющуюся палитру, то очень скоро убедитесь, что ее внешний вид окажется несколько изменен. Это может стать величайшим образцом поп-арта, но вряд ли будет пригодно для вашей игры. Чтобы обойти такое затруднение, необходимо создать палитру, включающую цвета, которые присутствуют в изображении.

Если вы намерены использовать в вашей игре фотографии и другие оцифрованные изображения, набора в 256 цветов хватит ненадолго. Запомните несколько правил:

- Подбирайте изображения, имеющие сходную цветовую палитру, причем стремитесь, чтобы она была по возможности ближе к той, которую вы хотите использовать в своей игре. Это правило позволит вашим картинкам отображаться более точно;
- Выбирайте изображения достаточно контрастные и в то же время с непротиворечивым цветовым решением. Если вы возьмете изображение, светлое с одного края и темное — с другого, это создаст лишние сложности в использовании принципа мозаичности, необходимого для трехмерных игр. Мы обсудим способы получения мозаичных изображений чуть позже;
- Не берите изображения с большим количеством мелких деталей. К примеру, у вас есть фотография, выполненная с высоким разрешением, изображающая компанию людей. Когда это изображение уменьшится до размеров 64x64 пикселя, оно, скорее всего, будет сильно отличаться от оригинала. Лица людей на нем могут оказаться представленными всего одним пикселем, а то и вовсе пропадут!

Вероятно, хорошей идеей будет отмасштабировать ваши фотографии или другие оцифрованные изображения до окончательного размера, в котором они предстанут в игре. Изменять масштаб

лучше всего перед тем, как вы начнете оптимизировать палитру. На то есть две причины:

- Во-первых, после изменения масштаба пропадет часть цветов, что облегчит оптимизацию палитры;
- Во-вторых, когда придет время собирать разрозненные изображения единый PCX-файл с общей палитрой, этот процесс пойдет намного легче.

В зависимости от того, используете ли вы 256-цветный или 24-битный графический редактор, нужно по-разному подходить и к созданию окончательной 256-цветной палитры.

Как можно скомбинировать палитры нескольких картинок без ухудшения изображения при использовании 256-цветной программы рисования? Один подход состоит в том, чтобы уменьшить количество цветов, насколько это возможно без ущерба для качества.

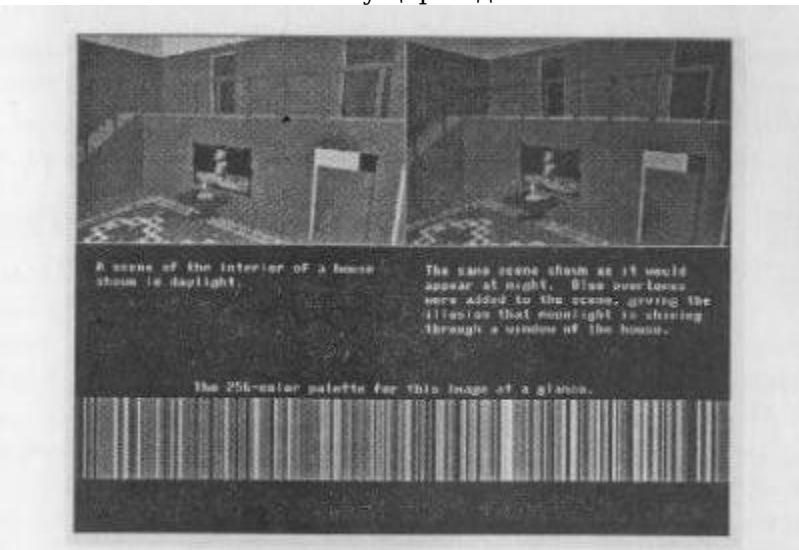


Рис. 16.2. *Палитра цветов, расположенных «по умолчанию», как это принято в графическом файле (EXAMP02.PCX).*

Позвольте мне продемонстрировать это на примере. Взгляните на рисунок 16.2. Слева изображена комната, как она выглядит при дневном освещении, а справа показан тот же самый вид ночью. Рисунок 16.3 ничем не отличается от предыдущего за исключением того, что палитра перекомпонована по градиентам, Сохранение цветов, уже имеющихся в изображении, упростит создание трехмерного изображения от руки.

Рис. 16.3. *То же самое изображение, что на рисунке 16.2, но с палитрой, перекомпонованной по градиентам (EXAMP03.PCX).*

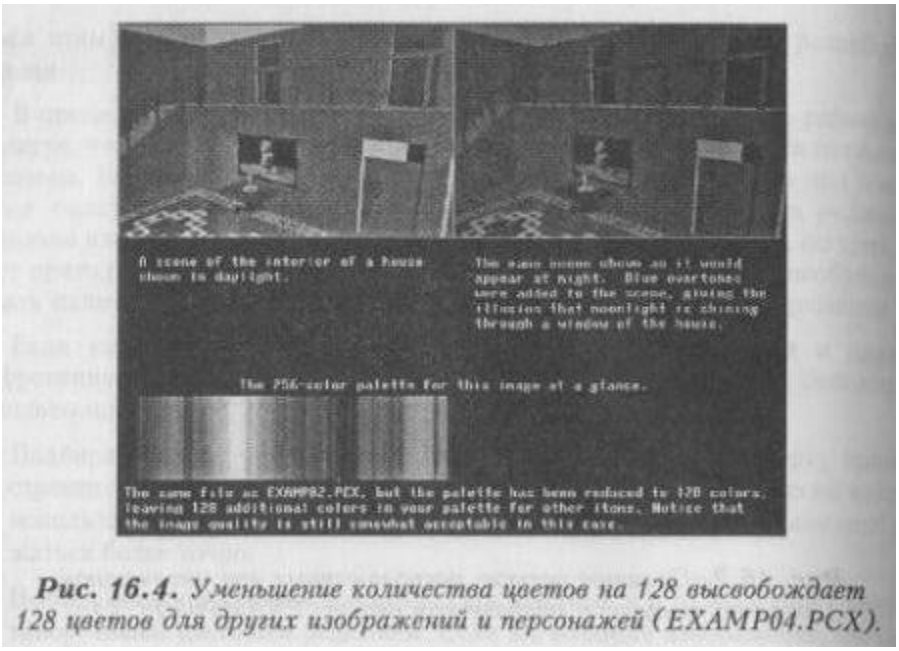


Рис. 16.4. *Уменьшение количества цветов на 128 высвобождает 128 цветов для других изображений и персонажей (EXAMP04.PCX).*

Теперь посмотрим на рисунок 16.4 (файл EXAMP04.PCX на дискете). В этом файле количество цветов, содержащихся в изображении, может быть уменьшено до 128. Это приведет к небольшому ухудшению качества изображения, но не сильно испортит общий вид рисунка. Зато у вас высвободится половина цветов, которые могут быть использованы для других картинок.

Посмотрим, что произойдет, если мы уменьшим количество цветов еще наполовину, то есть до 64. Взгляните на рисунок 16.5 (файл EXAMP05.PCX). Качество изображения снова ухудшилось и теперь даже может потребоваться слегка подкрасить его вручную. Сокращение палитры можно продолжать и дальше, и только вы решаете, остается ли качество все еще достаточно приемлемым для вашей игры.

Подобным образом нужно обработать все изображения, которые вы хотите ввести в игру. Но не забывайте, что общее количество цветов после объединения всех рисунков не может превышать 256.

Следующим шагом было бы неплохо собрать все изображения, палитр которых вы сократили, в одном графическом файле. Некоторые программ позволяют объединять палитры нескольких изображений в одной, и вы можете использовать ее для разработки оставшихся картинок вручную.

В 24-битовом графическом редакторе несколько проще создать новое изображение, включающее в себя все или почти все собранные вами промасштабированпые картинки. Например, на экране с разрешением 640x480 точек без труда можно разместить от 50 до 60 изображений размером 64x64 пикселя, даже если оставлять между ними для большей наглядности некоторый зазор. Кроме того, работая с 24-битовой программой, вам не нужно беспокоиться насчет сжатия палитры каждого изображения до нескольких цветов. Вы можете поместить все картинки в один файл, а затем просто отконвертировать его в 256-цветное изображение. Программа сама отберет для палитры те цвета, которые наиболее часто встречаются в изображениях.

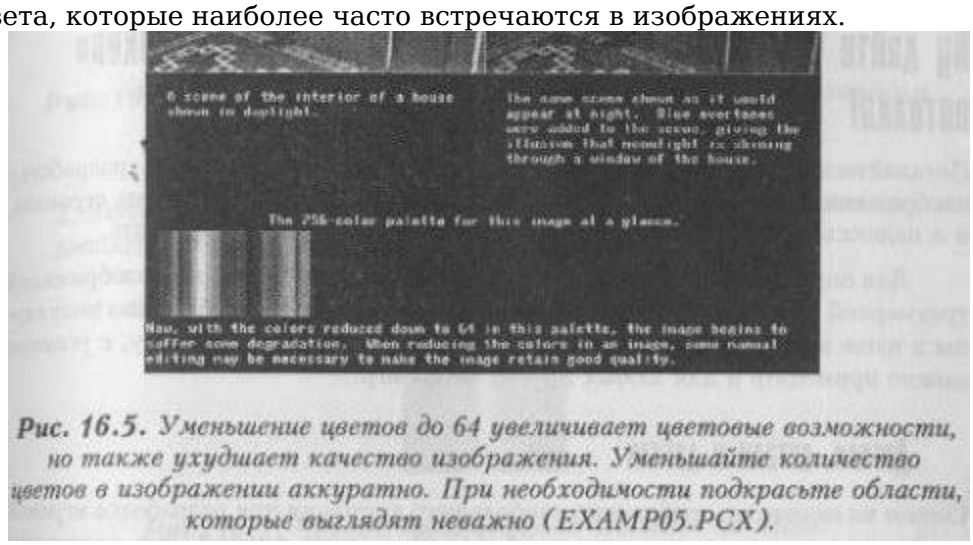


Рис. 16.5. *Уменьшение цветов до 64 увеличивает цветовые возможности, но также ухудшает качество изображения. Уменьшайте количество цветов в изображении аккуратно. При необходимости подкрасьте области, которые выглядят неважно (EXAMP05.PCX).*

Специальные соображения по поводу цифрового видео

Цифровые видеокадры значительно чаще используются для представления персонажей, чем образы, нарисованные от руки. С приходом Microsoft Video for Windows объединение оцифрованных видеофрагментов в играх и другом программном обеспечении является вполне обычным делом. Если ваша игра должна включать цифровые видеопоследовательности, убедитесь, что в каждый кадр вставляется одна и та же палитра. Для выполнения этой задачи у вас должны иметься соответствующие инструментальные средства. Усечение цветов в цифровой видеопоследовательности — нелегкое дело, особенно если отснято достаточно большое количество кадров. Обычно неплохо транслируются последовательности типа неподвижной «говорящей головы», потому что цвета в каждом кадре довольно хорошо согласуются между собой. Также можно снимать персонаж, расположив его перед синим экраном и осветив его так, чтобы на фон не падало никаких теней. Затем образ персонажа может быть наложен на любой другой фон с помощью специального технического приема, называемого Chromakeing (хромакей). Такой прием обычно используется в телевизионных передачах, например, для совмещения изображения ведущего с картами погоды (более подробно об этой процедуре говорилось в восьмой главе, «Высокоскоростные трехмерные спрайты»),

Ну дайте мне место (разработка стен, пола и «кирпичиков» потолка!

Догадайтесь, о чем мы сейчас будем говорить! Мы готовы начать разработку изображений прямо сейчас. К этой части вы должны отнестись очень серьезно и я надеюсь преподать достаточно основ, чтобы вы смогли стартовать.

Для определенности давайте сконцентрируемся на разработке изображений трехмерной приключенческой игры, поскольку такие игры необычайно популярны в наше время. Но те же самые принципы, о которых я расскажу, с успехом можно применять и для любых других типов игр.

Детали, детали и детали

Одним из моментов, требующих наибольшего внимания при разработке игровой графики, является включение в изображение мелких деталей. Это мастерство приходит с практикой и ему стоит поучиться. Например, голая каменная стена выглядит не слишком убедительно, но если на ней подрисовать трещины, пучки мха и другие незначительные подробности, это заметно улучшит ее внешний вид. Кроме того, такие дополнительные мазки придадут окончательному виду вашей игры большую индивидуальность. Запомните: каждый, даже самый мелкий штрих приоткрывает окно в ваш воображаемый мир, и вы должны заставить поверить игроков, что они действительно в него попали!

Разработку стен, пола и «кирпичиков» потолка лучше проводить в увеличенном масштабе изображения, используя команду ZOOM. Это полезно по двум причинам:

- При добавлении мелких деталей это помогает точно выбирать те пиксели, которые нужно изменить;
- Кроме того, вы сразу же видите, как «кирпичики» выглядят вблизи и при необходимости сможете вносить изменения, чтобы улучшить их внешность.

Давайте шаг за шагом рассмотрим этапы создания «кирпичика» стены. Эти шаги изображены в файле EXAMP06.PCX.

1. Начнем с того, что с помощью горизонтальных линий, изменяющих свой цвет от темносерого до почти белого и наоборот, изобразим прямоугольник размером 64x64, как показано на рисунке 16.6. Плавный переход цветов создает иллюзию того, что панель управления изготовлена из нержавеющей стали или алюминия.

Рис. 16.6. Прямоугольник 64×64, нарисованный с плавным переходом цвета от серого к белому.

2. Затем нарисуем черный прямоугольник, в котором будет располагаться дисплей панели управления (рис. 16.7).

Рис. 16.7. *Черный прямоугольник вставлен на место дисплея панели управления.*

3. Следующим шагом обведем черный прямоугольник линией медного цвета (рис. 16.8).

Рис. 16.8. *Бордюр медного цвета шириной в 1 пиксель, ограничивающий черный прямоугольник.*

4. Чуть ниже дисплея панели управления расположим несколько линии темного (25% черного) цвета, как вы можете видеть из рисунка 16.9. Это слегка оттеняет серый тон, делая верхнюю часть изображения более привлекательной.

5. На пятом шаге добавим детали панели управления, показанные на рисунке 16.10. Мы нарисуем красные и желтые яркие пиксели, изображающие светодиоды, яркокрасные цифры и яркосинюю кривую, напоминающую осциллограмму.

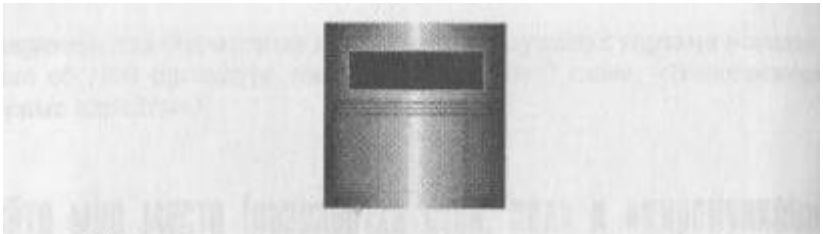


Рис. 16.9. *Три темных линии, дорисованные в верхней части, улучшают качество изображения.*

Рис. 16.10. *Светодиоды, цифры и другие детали, добавленные в область дисплея панели управления.*

6. Теперь, когда верх экрана проработан, нижняя часть стала выглядеть еще более пустой, так что надо ее чем-то заполнить. Для создания ощущения трехмерности добавим несколько серых теней расцветим их вертикальным градиентом. Эти изменения отображены на рисунке 16.11.

Рис. 16.11. *Место, освобожденное для выключателя.*

7. Затем оттеним отведенную для выключателя область светлыми и темными тонами, чтобы придать ей выпуклый вид (рис. 16.12).

8. Наконец, нарисуем сам выключатель, выполненный в медных тонах. вот мы и получили лицевую сторону панели управления! Окончательный результат показан на рисунке 16.13.

Для сооружения стен изготовим несколько новых "кирпичиков", внеся •некоторые изменения в уже нарисованный. Затем, используя премы вырезания и приклеивания, размножим созданные «кирпичики» и выстроим их в ряд,

Рис. 16.12. *Область выключателя, оттененная светлыми и темными тонами серого.*

Рис. 16.13. *Заключительная дорисовка: выключатель медного цвета добавлен в изображение панели управления.*

чтобы убедиться в том, что соседние области на стыках гармонируют друг с другом. Это позволяет избежать противоречий в соединении графических элементов, появляющихся в игре, а также помогает обойтись без швов при выводе немозаичных участков изображения. Повторю, что шаги, приведенные здесь, продемонстрированы в файле EXAMP06.PCX.

Придание «кирпичикам» стен глубины трехмерного пространства

Созданные вами «кирпичики» могут выглядеть на мониторе абсолютно безупречно, но когда вы помещаете их в трехмерное пространство игры, они вдруг становятся выглядящими как-то не так. В чем же дело?

Как правило, когда «кирпичики?» в трехмерном пространстве рассматриваются под углом, участки, имеющие темный оттенок, кажутся утопленными внутрь, и наоборот, светлые фрагменты выдвигаются на передний план. Помните это, когда создаете изображения и старайтесь правильно использовать. Чтобы придать вашим «кирпичикам» глубину и форму, воспользуйтесь приемом, продемонстрированным на рисунке 16.1. Постепенно изменяйте оттенки цветов от светлого к темному, располагая светлые ближе к поверхности, а темные - в глубине. Иногда, чтобы придать определенным частям более реалистичный вид, используйте резкий переход между оттенками светлого и темного,

Масштабирование изображения

Существует пара моментов, связанных с изменением масштаба изображений, используемых в качестве «кирпичиков», о которых необходимо упомянуть.

Рис. 16.14. Сцена, созданная в Vista Pro (EXAMP13.PCX).

Поговорим о создании сцены дальнего плана в программе создания пейзажей наподобие Vista Pro. Предположим, вам нужно изменить масштаб полученного изображения так, чтобы можно было использовать нарисованные горы в качестве «кирпичиков» пейзажа. Помните, что изначально ваше изображение имеет размер 640x480 точек. В верхней части картинки (рис. 16.14) показан кусок неба, а в нижней - трава. Нас же в основном интересует центральная область, занятая горами и холмами.

Только не вздумайте использовать все это изображение как один «кирпичик» размером 64x64, иначе горы после изменения масштаба окажутся сжатыми также и по горизонтали. В этом случае имеет смысл создать несколько «кирпичиков», разбив рисунок на части, и затем последовательно поместить их на экран.

Если вы хотите, чтобы окончательное изображение размещалось в двух кирпичиках 64x64, вы должны подстричь исходную картинку до 640x320 точек (чтобы ширина оказалась ровно в два раза больше высоты). Для трех кирпичиков нужно вырезать область размером 640x213 точек (высота равна 1/3 ширины), а для четырех - 640x160 (высота равна 1/4 ширины).

Существует еще несколько возможностей улучшить общее впечатление. Когда будете вырезать изображение, не устанавливайте высоту сразу пикселей, а захватите кусочек побольше, потому как вам может захотет сохранить немного неба или травы. Оставляйте немного дополнительных деталей, чтобы поиграть с ними во время предварительного отсечения, потому если из изображения удалить траву и небо, оно будет выглядеть со по-другому. На рисунке 16.15 отмечена область, которая в нашем примере будет удалена, а рисунок 16.16 показывает окончательное изображение размер 640x160 точек.

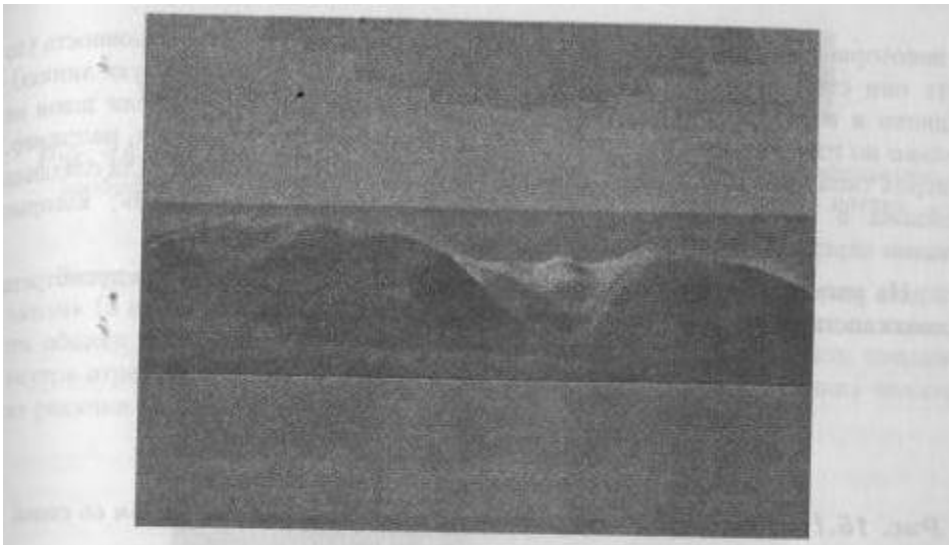


Рис. 16.15. *Подсвеченные области на этой фотографии будут удалены с изображения. Окончательный размер картинки будет 640×160 пикселей (EXAMP12.PCX).*

Рис. 16.16. *Окончательное изображение 640×160, полученное после всех отсечений (EXAMP11.PCX).*

Следующим шагом нужно уменьшить изображение с 640x160 до 256x64 (до размера четырех "кирпичиков", размещенных один за другим). Многие графические программы позволяют масштабировать изображения. Некоторые из них не разрешают непосредственно указывать новую ширину и высоту изображения или задавать масштаб в процентном соотношении. Чтобы определить эту велечину, разделите ширину изображения, которую нужно получить (в нашем случае 256) на исходную ширину (640) и умножьте на 100. В нашем примере процентное соотношение размеров оригинального и результирующего изображении получается равным 40.

Создание бесшовных мозаичных изображений

К бесшовным относятся те изображения, которые можно разместить один за другим так, что не будет заметно, где заканчивается одно и начинается другое.

В некоторых играх изображения имеют только горизонтальную бесшовность (то есть они стыкуются друг с другом в непрерывную горизонтальную линию) Однако в некоторых играх требуется, чтобы изображения не имели швов не только по горизонтали, но и по вертикали. Это может понадобиться, наприме, в играх типа имитаторов полета для составления карты местности. Для создания пейзажа в таких случаях используются бесшовные «кирпичики», которые плавно переходят один в другой и по горизонтали и по вертикали.

На рисунке 16.17 показано, что получается, если заранее не предусмотреть возможности бесшовного соединения картинок.

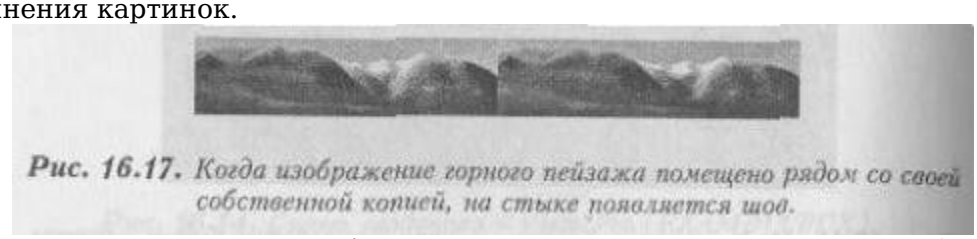


Рис. 16.17. *Когда изображение горного пейзажа помещено рядом со своей собственной копией, на стыке появляется шов.*

Здесь наша горная цепь, отмасштабированная до размера 256x64 точек (размер четырех кирпичиков) помещена вслед за своей копией. Обратите внимание, что в месте соединения изображений появился хорошо заметный шов:рисунки не перетекают плавно один в другой. Этот шов необходимо отредактировать так, чтобы объединенные изображения выглядели как одна непрерывная картинка.

Существует два основных подхода к созданию бесшовных «кирпичиков». Во-первых, что вполне применимо к данному примеру, можно сделать зеркальную копию исходного рисунка. После этого

изображения хорошо состыкуются друг с другом и их можно затем показать в игровом пространстве, как на рисунке 16.18.

Рис. 16.18. *Один метод создания бесшовного «кирпичика»: сделать зеркальную копию и поместить ее следом за исходным изображением.*

Однако этот метод будет работать не для всех изображений. В тех случаях, когда вы хотите расположить рядом два совершенно одинаковых «кирпичика». потребуется отредактировать изображение в местах соединений. Подбирая цвета из области, окружающей шов, исправьте рисунок так, чтобы переход от одного изображения к другому не был заметен. На рисунке 16.19 показано, как выглядит соединение после модификации изображений.

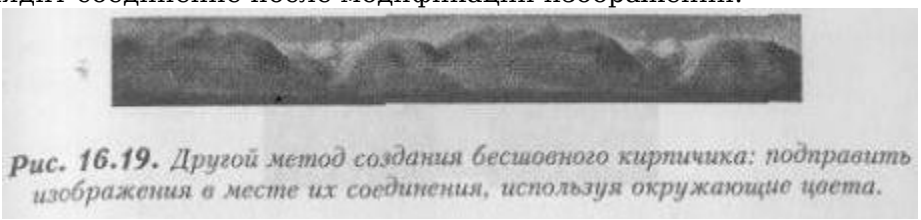


Рис. 16.19. *Другой метод создания бесшовного кирпичика: подправить изображения в месте их соединения, используя окружающие цвета.*

Затем новое изображение, совпадающее по размерам с исходным «кирпичиком» (в нашем случае 256x64 пикселя), вырезается откуда-нибудь из середины общего изображения. Убедитесь, что отредактированная область попадает внутрь отрезаемой части. Полученный в итоге бесшовный «кирпичика показан на рисунке 16.20.

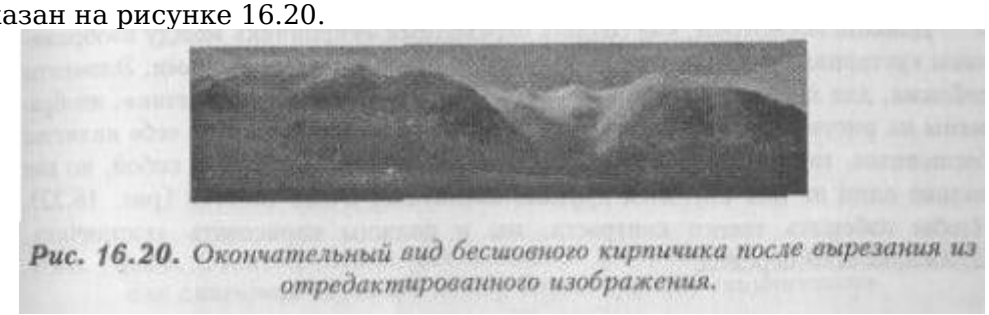


Рис. 16.20. *Окончательный вид бесшовного кирпичика после вырезания из отредактированного изображения.*

Приведенные выше иллюстрации можно увидеть в цвете, просмотрев файл EXAMPЮ.PCX.

Особые ситуации

Имеются еще и другие соображения, которые нужно принимать в расчет при создании «кирпичиков» стен. Иногда к их разработке требуется применять особый подход, чтобы воспроизвести определенные эффекты, например, открытие и закрытие дверей. Вдобавок, при переходе игрока из одного помещения в Другое может потребовать создание промежуточного «кирпичика», показывающего торец дверного проема. Кроме того, можно заставить некоторые из «кирпичиков» изменять свой внешний вид, что сделает путешествие более Интересным. В данном разделе обсуждаются эти типы ситуаций.

Переходы от одного кирпичика к другому

Декорации хотя бы иногда должны меняться, иначе игра будет довольно скучной. Для перехода от одного пейзажа к другому нужно научиться создавать связующие «кирпичики», чтобы смена окружения происходила плавно.

Предположим, у вас имеется пейзаж, представляющий собой заросли кустарника, а погом вы оказываетесь, скажем, в пещере. И если, вы не хотите,

Рис. 16.21. *Эти «кирпичики», будучи помещенными бок о бок со своей копией, выглядят бесшовными.*

чтобы переход от одного типа «кирпичиков» к другому слишком сильно бросался в глаза, нужно изготовить дополнительный «кирпичик», связывающий их в непрерывную последовательность.

Давайте посмотрим, как создать переходный «кирпичик» между изображением кустарника и

элементом стены пещеры, выложенной валунами. Элементы пейзажа, для которых мы собираемся создать переходный «кирпичик», изображены на рисунке 16.21. Как видите, каждый «кирпичик» сам по себе является бесшовным, то есть они прекрасно соединяются каждый с самим собой, но как только один из них сменится другим, шов станет очень заметен (рис, 16.22). Чтобы избежать такого контраста, мы и должны нарисовать «кирпичик», сглаживающий переход.

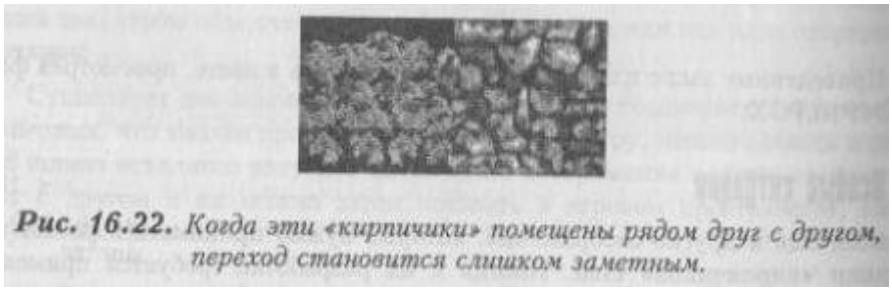


Рис. 16.22. *Когда эти «кирпичики» помещены рядом друг с другом, переход становится слишком заметным.*

Идея переходных «кирпичиков» состоит в том, что с одного края он плавно переходит в один «кирпичик», а с противоположной стороны — в другой. Скажем, вы хотите создать промежуточный «кирпичик», связывающий элементы кустарника и пещеры. Для этого нужно выполнить следующие шаги:

1. Разбейте обе картинки точно пополам. Затем возьмите левую половину изображения кустарника и соедините ее с правой частью «кирпичика» пещеры. Теперь левый край полученного рисунка состыкуется без шва с «кирпичиком» кустарника, а правый — с «кирпичиком» пещеры (рисунок 16.23).

2. Отредактируйте место соединения половинок изображений так, чтобы убрать с рисунка шов. Для достижения плавного перехода в этом случае достаточно сгладить Очертания кустарника и валунов пещеры, заполнив

Рис. 16.23. *Левая половина «кирпичика» кустарника, помещенная напротив правой половины «кирпичика» пещеры.*

пространство между ними цветом фона. После этого можно смягчить контраст между фоном и изображением, воспользовавшись соответствующим инструментом (рис. 16.24).

Рис. 16.24. *Область соединения половинок изображений отредактирована для смягчения перехода между двумя типами «кирпичиков».*

Аналогичным образом можно изготовить промежуточный «кирпичик» для перехода от изображения пещеры к кустарнику, только в этом случае нужно соединить левую половинку фактуры пещеры с правой половинкой фрагмента кустарника. После этого у вас появится плавный переход от одной сцены к Другой независимо от последовательности смены декораций. Рисунок 16.25 показывает, как выглядит переходный «кирпичик», который мы только что создали, когда он помещен в надлежащее место игрового пространства.

Двери и лифты

Раздвижные двери и двери лифтов состоят из двух «кирпичиков»: собственно Двери и косяка. Для этого примера возьмем два «кирпичика», которые были

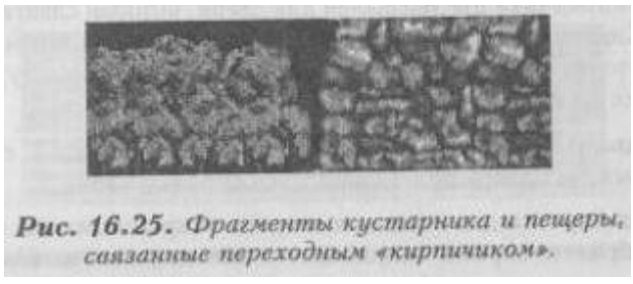


Рис. 16.25. *Фрагменты кустарника и пещеры, связанные переходным «кирпичиком».*

Рис. 16.26. *Два типа «кирпичиков», выбранные для создания лифта и разработанные с использованием уже созданного «кирпичика» панели управления.*

созданы, когда изготовлялась область панели управления. Эти два изображения исключительно хорошо подходят для двери лифта. «Кирпичик» справа безупречно подходит для косяка, а кирпичик слева будет прекрасен для двери лифта (это показано на рисунке 16.26).

Надо подумать о «кирпичике», который будет представлять изображение В проеме открывающейся двери. Чтобы сделать Косяк и дверь из этих «кирпичиков» надо:

1. Нарисовать темный прямоугольник в центре «кирпичика»;
2. Немного смягчить и подровнять границы, чтобы линии не выглядели отрывистыми;
3. Добавить небольшие детали, такие как светодиоды или кнопки. Результат показан на рисунке 16.27.

Рис. 16.27. *Первый кирпичик, показанный на рисунке 16.26, модифицирован для создания дверного косяка.*

Вам необходимо иметь два «кирпичика» для двери лифта;

• Первый «кирпичик» предназначен для двери, которая сдвигается влево или вправо. Кнопки на этом «кирпичике» расположены на левом краю: сторона двери, которая будет видимой, пока дверь уберется в стену (как показано на рисунке 16.28);

• Второй дверной «кирпичик» имеет кнопки на правой стороне двери. Открываясь, эта дверь будет сдвигаться справа налево.

Во многих случаях для создания второго «кирпичика» вы можете только зеркально отобразить первый «кирпичик» и внести в него изменения.

Рис. 16.28. *Два дверных «кирпичика» — один для сдвига влево, а другой для сдвига вправо, созданные из левого «кирпичика», показанного на рисунке 16.26.*

Специальные эффекты и движущиеся стены

В некоторых кадрах отдельные «кирпичики» могут изменять свой внешний вид, что делает общую картину более привлекательной. Например, по стенам можно развесить пылающие факелы, установить в комнате камин или расставить панели управления.

Один из «кирпичиков» панели управления мы уже изготовили ранее и теперь можем попытаться оживить его. Давайте посмотрим, что для этого нужно сделать. Взгляните на следующие шесть элементов, три из которых показаны на рисунке 16.29, а остальные — на рисунке 16.30. Заметьте, что каждый «кирпичик» слегка отличается от других. Некоторые светодиоды, зажженные на одной панели, погашены в других местах, различаются также и цифры на дисплее, и синяя линия осциллограммы. На последних трех «кирпичиках» выключатель изменяет свое положение от верхнего до нижнего, что может быть использовано для изображения работающей или отключенной панели управления.

Рис. 16.29. Дисплей панели управления немного отличается в каждом из этих трех «кирпичиков».

Рис. 16.30. На этих трех панелях выключатель движется сверху вниз.

Будучи оживленными, эти «кирпичики» выглядят так, словно панель управления работает: хаотично вспыхивают светодиоды, что-то мелькает на дисплее. Чтобы изготовить такие «кирпичики», достаточно сделать несколько копий исходного изображения, а затем слегка модифицировать каждую из них.

Удаление неровностей

Рассматривая «кирпичики» в трехмерном пространстве под углом, вы иногда можете обратить внимание, что некоторые области имеют сильные неровности Чтобы избежать этого, можно рисовать каждую горизонтальную линию шириной два пикселя, используя сглаживание вокруг каждой из них, чтобы сделать их более аккуратными. Иногда таким способом не удается избавиться от неровностей, особенно на маленьких объектах или спрайтах. В этом случае иногда помогает исправить ситуацию правильный подбор цветов.

Часто помогает выбор в качестве фонового цвета для спрайта того тона который преобладает в изображении, с последующим сглаживанием между основным рисунком и фоном. Затем оставшийся фон может быть заменен на «прозрачный». Это сложный процесс, но по мере накопления опыта он станет проще.

Как избежать цветовых проблем во время игры

Если вы заметите, что при удалении объекты выглядят темнее, чем вблизи, то может понадобиться отказаться от использования самых темных оттенков в каждом цветовом диапазоне. Однако не убирайте их из палитры совсем, потому как по ходу игры они могут все-таки понадобиться. Некоторые алгоритмы отрисовки игрового экрана заменяют оттенки цвета, использованные в ваших «кирпичиках», оттенками, зарезервированными для осуществления эффекта затенения. Если более темный оттенок отсутствует в палитре, VGA заменит его цветом, имеющим наиболее близкое значение RGB, что может оказаться крайне нежелательным.

Если б только у меня было оружие и еда! (разработка объекта)

На свете не существует игр, в коих отсутствуют какие-либо предметы, которые нужно найти, подобрать и как-то использовать во время вашего приключения. Должно быть, это трудная задача — драться с атакующими вас чудовищами, не имея ни оружия для отражения нападения, ни еды для пополнения сил. Кроме того, голые стены без малейшего намека на какую-либо обстановку также будут выглядеть довольно уныло, поэтому комнаты желательно заполнить какими-нибудь объектами.

Разработка объектов похожа на создание «кирпичиков» стен с одной только существенной разницей: в изображениях обязательно будут присутствовал невидимые или прозрачные области. .Каждый «кирпичик» стены представляет собой прямоугольник, все пиксели которого обычно отображаются на экране. Правда в некоторых играх встречаются перегородки с прозрачными областями. Это могут быть различные ворота, решетки, окна. Такие же «кирпичики» используются и для изображения открывающихся и закрывающихся дверей.

В процессе разработки объекты также представляются спрайтами прямоугольной формы, но те части, которые не должны быть видимы, заполняются «прозрачным» цветом. Поскольку «прозрачные» пиксели на экран не выводятся то сквозь эти участки будет виден находящийся под объектом фон.

Соотношение размеров объектов и стен

При разработке объектов их создают такими, чтобы на фоне стен они выглядели достаточно пропорционально. Учитывая, что средняя высота стены равна примерно 8 футам, а «кирпичики» стен имеют размер 64 пикселя, можно подсчитать, что на каждый фут приходится по 8 пикселей (64 пикселя разделить на 8 футов). Это правило хорошо работает для больших объектов, таких как мебель или стволы деревьев, однако для мелких предметов оно не выполняется. Было бы очень непросто нарисовать пистолет, имеющий размер всего около 8 дюймов, впихнув его в изображение высотой 5 или 6 пикселей. В этом случае можно применить некоторую художественную вольность. Просто рисуйте объекты такими маленькими, насколько это возможно, чтобы можно было различить детали, и не обращайте внимания на пропорции.

Помните о перспективе

Обратите внимание, что когда вы гуляете по коридорам вашего трехмерного пространства, поверхность земли или пола располагается немного под углом к горизонту. Внесение небольшого искажения в форму «кирпичиков» может существенно улучшить реалистичность восприятия. Если вы сделали своего персонажа достаточно высоким, чтобы он мог смотреть поверх стен, то можно изобразить открывающийся перед ним вид под несколько большим углом к горизонту. Однако постарайтесь не переборщить с перспективой, иначе это не будет согласовываться с остальным окружением.

Подайте сюда врагов! (разработка персонажей)

Теперь, когда игровое пространство и объекты созданы, вы должны вдохнуть жизнь в выдуманных персонажей, чтобы с ними можно было как-то взаимодействовать. Разработка персонажей, невидимому, является наиболее утомительной частью создания игры. Для этого придется неоднократно рисовать, по крайней мере, четыре, а то и восемь кадров для представления различных фаз движения существа. Каждый из этих кадров должен быть тесно связан с остальными, чтобы движения не выглядели дерганными.

Существует ли легкий путь?

Проще всего создавать персонажей и просматривать полученный результат с помощью какой-нибудь трехмерной анимационной программы типа 3D Studio Моделирование, анимация и последующее редактирование персонажей иногда может быть выполнено и вручную, но перспектива и оттенки персонажей при использовании анимационной программы окажутся более реалистичны. Если у вас есть только двухмерный графический редактор, то ничего не поделаешь придется каждый кадр рисовать вручную. Однако существует несколько технических приемов, позволяющих немного упростить задачу, и сейчас мы их обсудим.

Пропорции

Пропорции человеческого тела различны в зависимости от возраста. Что касается тела взрослого человека, то оно делится на 8 частей:

- Голова занимает одну восьмую часть от общей высоты тела. Этот раздел заканчивается в нижней точке подбородка;
- Ко второму и третьему разделам относятся шея и торс, причем третья часть заканчивается на уровне талии;
- Четвертый раздел — это область от талии до бедер;
- Пятая и шестая части содержат верхний отдел ног. Сюда же относятся и колени, находящиеся чуть выше нижней границы шестого уровня;
- Седьмой и восьмой разделы включают в себя нижнюю часть ног;
- Что касается рук, то локти располагаются немного выше уровня талии, а плечо — на стыке третьего и четвертого разделов. В целом рука начинается от плеча и заканчивается на уровне 4/5 пятого раздела.

Подробнее о структуре человеческого тела, пропорциях скелета и мускулатуры вы можете узнать из книг по анатомии. Иногда в играх и мультфильмах пропорции бывают несколько преувеличены. Удлиненные руки и ноги, гипертрофированная мускулатура и т, п. помогают придать персонажу более свирепую или героическую внешность.

Основополагающие принципы для оживления персонажа

Файл EXAMP15.PCX, изображенный на рисунке 16.31, включен для того, чтобы помочь вам в разработке идущего человекоподобного персонажа, рассматриваемого с восьми точек. Эти кадры были созданы с помощью 3D Studio и точно показывают, как выглядит персонаж с различных точек зрения. Для каждого вида изготовлено по четыре кадра. После оживления картинок движения героя выглядят вполне плавными и убедительными.

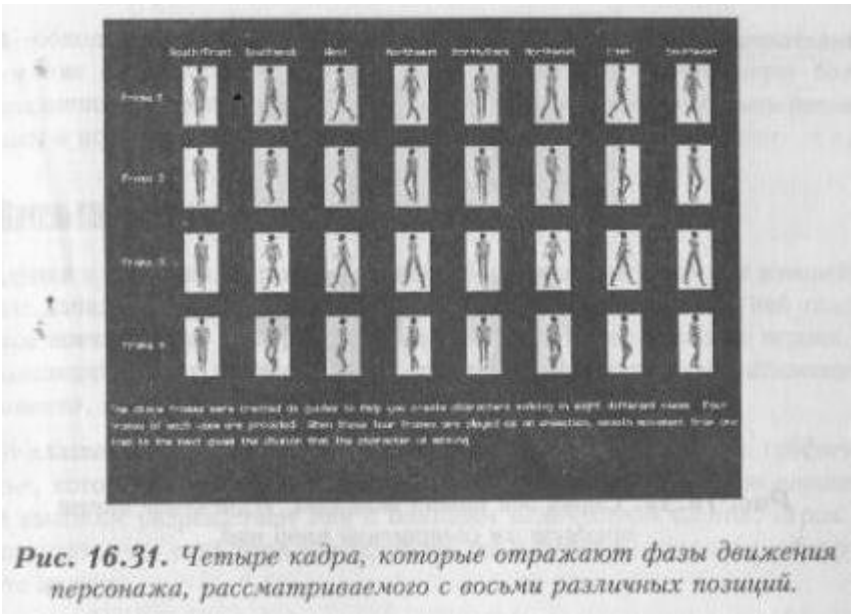


Рис. 16.31. *Четыре кадра, которые отражают фазы движения персонажа, рассматриваемого с восьми различных позиций.*

Технические приемы анимации

Внимательно изучив рисунок 16.31, вы можете заметить, что определенные части каждого кадра остаются относительно неподвижными. Положение головы и туловища человека не изменяются от кадра к кадру. Чтобы оживить персонаж было несколько легче, надо предпринять следующие действия:

1. В первую очередь нарисуйте области, остающиеся неизменными,
2. Затем сделайте необходимое количество копий кадра.
3. Наконец, дополните каждый кадр недостающими движущимися частями.

Этот прием дает вам одно преимущество: он помогает правильно расположить изображения фаз движения одно за другим. Мультипликация базируется иа неподвижных частях изображения и, выводя полученные таким способом кадры, вы можете получить достаточно плавное движение. Помните, что сдвиг неподвижной части изображения даже на один пиксель может привести к ситуации, когда движения вашего персонажа будут выглядеть дерганными.

Чудовища и прочие кровожадные персонажи

Стоит ли говорить, что, рисуя чудовищ для игры, можно отбросить в сторону всякие правила и забыть о соблюдении пропорций. Некоторые из ваших монстров, скажем, могут иметь головы в половину всего тела! Единственный принцип, которым нужно руководствоваться при их создании - это богатое воображение. Такое занятие больше похоже на развлечение: вам не нужно беспокоиться на счет реализма, ибо никто в действительности не знает, как на самом деле должен выглядеть предполагаемый монстр!

Рис. 16.32. Снова два наших монстра. Изменение цвета придает им совершенно иной вид.

Вокруг имеется обилие источников, которые могут подогреть вашу фантазию. В ближайшей библиотеке наверняка найдется фантастическая литература с иллюстрациями и в видеопрокате можно подобрать подходящий фильм. Комиксы также могут послужить хорошим источником идей. Изображение чудовищ и демонов в искусстве прошло длинный путь, так что без особого труда можно подыскать на эту тему работы художников многих столетий. Но помните: делайте ваших монстров как можно более оригинальными, чтобы ненароком не нарушить авторских прав.

Файл EXAMP16.PCX показывает несколько кадров с одним и тем же монстром. Как видите, он выглядит совершенно по-разному, если изменять его цвет. Это говорит о том, что даже небольшие изменения могут в корне преобразить ваше создание! Возможно, в дальнейшем вы составите целую библиотеку изображений всевозможных чудовищ.

Последние дорисовки

Вы могли бы подумать, что ваша графическая разработка завершена, но существуют еще несколько экранов, которые должны быть созданы, Это экраны, с которыми будет взаимодействовать пользователь: заставки, которые открывают и зарывают игру, граничный экран, окружающий игровое пространство, оценочные экраны, которые дают знать игроку о ноложении дел в игре.

Необходимо все эти экраны делать как можно более привлекательными. Следование этому принципу придаст внешнему виду вашей игры больший профессионализм. Ключевое правило здесь - это стремление быть последовательным в подборе цвета и создании настроения.

Начальная и конечная заставки

Начальная и конечная заставки, вероятно, будут самыми важными в вашей игре, причем начальная заставка всегда имеет большее значение. В ней создается первое впечатление» от игры, которое засядет в голове у вашего игрока. Если открывающий экран нарисован так, что успешно притягивает внимание, то, несомненно, и игра в целом привлечет внимание.

Желательно разрабатывать начальную заставку в том же графическом режиме, который используется и в игре. Если открывающий экран нарисован с более высоким разрешением или с большим количеством цветов, игрок будет предполагать, что сама игра выглядит так же и будет разочарован, обнаружив, что это не так.

В начальной заставке вы можете:

- Дать более детализированное изображение главного героя игры;
- Показать основного врага, продемонстрировав весь его устрашающий потенциал и как бы говорящего: «Приди и попробуй меня победить»;
- Нарисовать несколько монстров из вашей игры в атакующих позах;
- Еще один неплохой выбор — использовать несколько видеокадров, выполненных в том же разрешении, что и сама игра.

Для конечной заставки может применяться та же техника. Главное назначение этой заставки — пробудить у игрока желание вернуться в игру.

Оформление оценочного экрана

Оценочный экран служит для отражения текущего положения в игре. Ниже приведены некоторые элементы этого экрана, которые могут быть в него включены:

- Зона для показа очков и уровня игры;
- Зона, которая показывает, какое оружие игрок в данный момент использует;

- Зона, отражающая состояние здоровья игрока;
- Зоны, демонстрирующие, сколько боеприпасов, пищи или медикаментов игрок подобрал по дороге;
- Карта местности, на которой показано местоположение игрока и некоторых важных предметов.

Экраны для отражения статуса и изменения конфигурации игры

Здесь приводятся несколько элементов, которые могут быть показаны в любой момент во время игры:

- Текущий статус уровня, который занимает игрок;
- Общий статус игры;
- Экран конфигурации игры;
- Экран помощи;
- Командный экран;
- Экран для выбора уровня сложности игры.

Анимация

Анимация также очень эффектна для начала и конца игры. Вы можете использовать мультипликацию для привлечения внимания к торговой марке вашей фирмы, для показа фабулы игры или для завершения игры с предварительным просмотром того, что происходило. Наиболее общим форматом для анимации является FLI- или FLC-форматы. Формат FLI имеет разрешение 320x200, что вполне достаточно для большинства условно-бесплатных игр. Формат FLC был разработан несколько позднее и поддерживает любое необходимое разрешение.

Кадры для привлечения внимания

Раз ваши персонажи, сцены, объекты и экраны интерфейса созданы и объединены в игре, вам может захотеться взять несколько экранных кадров, чтобы поместить их на ББС или аналогичную информационную службу для привлечения внимания к вашей игре. Существует немало коммерческих пакетов, предназначенных именно для этих целей. Вам только необходимо убедиться, что они корректно работают в выбранном графическом режиме. Другим способом является создание собственной программы для захвата и сохранения на диске кадров при нажатии определенной «горячей» клавиши.

ИТОГ

Поскольку сейчас происходит бурный рост мощности ПК, то появляется все больше игр со сложной анимацией. Поэтому как можно больше учитесь работать с графикой, видео и анимацией. Вы должны пройти долгий путь, прежде чем научитесь создавать высококачественные игры. Достигнув этого, вы получите величайшее наслаждение, когда узнаете, что люди всего мира получают удовольствие у вашей игры. Это станет определением вашего успеха, как разработчика игр.

ТЕХНИКА СОЗДАНИЙ ПАРАЛЛАКСА

Вы, конечно, не раз обращали внимание, глядя из окна автомобиля, что близлежащие объекты перемещаются гораздо быстрее, чем удаленные. Этот эффект получил пугающее название — параллакс. В повседневной жизни вы настолько часто наблюдаете это явление, что, скорее всего, считаете его чем-то само собой разумеющимся.

Параллакс является только одним из многих эффектов нашего визуального восприятия. Другой хорошо известный эффект - это перспектива. Перспектива и параллакс вкупе с другими ощущениями, такими как равновесие и слух, формируют завершенную картину окружающей среды.

Что же такое параллакс и каким боком он связан с программированием игр? Параллаксное смещение - это технический прием, применяемый в видеографике, когда два или более слоя графического изображения перемещаются в одну сторону, но с различными скоростями. Такое относительное смещение слоев обеспечивает некие визуальные эффекты, необходимые для реалистичной имитации глубины пространства и движения.

Еще кое-что о режиме 13h

Прежде чем заняться разработкой первой программы, реализующей параллакс давайте еще раз поговорим об избранном видеорежиме. Как мы уже обсуждали в пятой главе «Секреты VGA-карт», наиболее популярным видеорежимом для программирования игр является режим 13h. Причиной номер 1 является его совместимость с различными микросхемами VGA. Другая причина, вероятно заключается в простоте программирования. Все нижеследующее является кратким обзором приемов программирования VGA в режиме 13h. Для более глубокого обзора необходимо вернуться к пятой главе. С другой стороны, если вы в совершенстве овладели программированием VGA-карт, то без ущерба можете пропустить этот раздел.

Режим 13h поддерживает разрешение 320x200 пикселей при одновременном отображении 256 цветов, где каждый пиксель представлен одним байтом. Следовательно, видеобуфер имеет размер 64000 байт. Эти 64K отображаются в область памяти, начиная с адреса A000:0000.

Все, что требуется для изображения пикселя в режиме 13h, это записать в видеобуфер байт. Проще всего это сделать, используя следующий код:

```
char far *VideoMem =MK_FP(0xA000,0) ;
```

Чтобы изобразить отдельный пиксель в позиции (200,100) с цветовым значением равным 2, вам необходимо только рассчитать смещение байта относительно начала видеопамяти. Смещение вычисляется путем умножения Y-координаты на 320 (поскольку в строке 320 пикселей) и добавлением X-координаты:

```
PixelOffset = y * 320 + x;
```

После этого вы можете обращаться к видеопамяти как к элементам обычного одномерного массива:

```
VideoMem[PixelOffset] = 2;
```

Используя функцию memcpyO из стандартной библиотеки Си, можно разом вывести на экран целую строку пикселей. Эта функция является одним из быстрейших путей копирования данных из одного места памяти в другое. В некоторых компиляторах эти функции бывают встроенными, что значительно увеличивает скорость их выполнения.

Например, следующий фрагмент копирует 320 байт из массива Scr в видеобуфер:

```
memcpy(VideoMem, Scr, 320);
```

Поскольку VideoMem ссылается на начало видеопамяти, эта функция изображает одну полную строку пикселей на экране дисплея.

Вся графика сначала выводится в буфер, расположенный в системной памяти (в дублирующий буфер), а затем, когда кадр полностью сформирован, он копируется на дисплей единым блоком также с помощью функции memcpy().

Это все, что необходимо на данный момент времени знать по поводу режима 13h, так что двинемся дальше.

Примечание по поводу демонстрационной программы

Я привожу некоторые небольшие демонстрационные программы для иллюстрации технических приемов, описанных в этой главе- По указанным выше причинам все программы работают в видеорежиме 13h. Я старался сделать их максимально удобочитаемыми, поэтому написаны они не самым оптимальным образом и не слишком эффективны.

Внимание

Все программы в этой главе написаны на Borland C++ 3.1 и Турбо ассемблере 3.1. Однако все примеры на Си были написаны с максимальной осторожностью, без привлечения особенностей Borland C. Так что они должны легко компилироваться любыми трансляторами C/C++ без внесения значительных изменений. Программы на ассемблере писались с использованием ключа IDEAL. Обратите внимание, что они не используются в демонстрационном примере, приведенном в этой главе, поскольку здесь же приведены их аналоги на Си.

Первый шаг

Перед тем как мы сможем смещать несколько слоев графического изображения, нам нужен метод для смещения отдельного слоя графики- Несомненно, существует несколько различных путей для получения смещающейся графики. Первый метод, который мы обсудим, довольно прямолинеен и работает хорошо для несложного, повторяющегося смещения.

При повторяющемся смещении движущееся изображение, уходя за границу зкрана, тут же появляется с его противоположной стороны. Несмотря на то, что зта техника может показаться не слишком полезной, она наиболее проста в осуществлеиии и может быть использована неоднократно.

К примеру, фон в динамичных играх не привлекает к себе слишком большого внимания, так что пользователи, вероятно, даже и не заметят, что вид облачного голубого неба на заднем плане периодически повторяется.

Так как же получить на экране изображения и заставить его циклически возвращаться при достижении границ экрана? Один из простейших способов заключается в выводе битовой карты на экран двумя порциями. Начнем с тот что логически разделим наше изображение на две части.

Запомните положение логической границы, разделяющей изображение на две части. Полученное число будет соответствовать ширине левой части изображения. Теперь назовем это число LeftHalf и установим начальную ширину левой части равной одному пикселю.

Следовательно, логической шириной правой части является общая ширина изображения минус ширина правой части. Теперь вы рисуете логическую правую часть изображения в левой половине экрана и наоборот, изображаете левую половину изображения на правой стороне экрана. Взгляните на рисунок 17.1 на котором показан этот процесс.

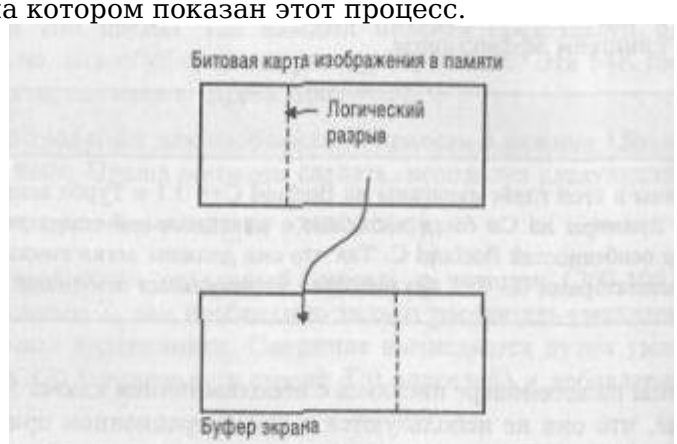


Рис. 17.1. *Непрерывно смещающееся изображение.*

Так каким же образом этот способ вывода изображения приводит к появлению эффекта смещения? Для понимания происходящего посмотрим, что случится, когда вы уменьшаете значение LeftHalf на единицу.

Декрементируя LeftHalf, вы уменьшаете на единицу ширину левой половины. В результате последний пиксель левой половины становится первым пикселем правой. Такое логическое перемещение данных изображения из одной половины в другую как раз и создает эффект прокрутки.

Затем обе части изображения выводятся на экран. Делается это так. Начиная с левой границы экрана, рисуется правая логическая часть изобраяжения:

- Сначала выводится строка пикселей изображения, начиная со столбца определенного переменной LeftHalf. Количество рисуемых писелей равно общей ширине изображения минус LeftHalf;
- Продолжайте рисование пикселей от левого края изображения до столбца LeftHalf.

Если положение логического разрыва изменить и перерисовать изображение заново, оно будет выглядеть движущимся по экрану.

Взгляните на следующий пример, демонстрирующий рисование двух половинок строки развертки изображения (вспомните, что разверткой называется полная горизонтальная строка пикселей).

```
    // нарисовать левую половину изображения
    memcpy(Screen+320-LeftHalf, Bitmap, LeftHalf) ;
    // нарисовать „правую часть изображения
    memcpy(Screen,Bitmap+LeftHalf,320-LeftHalf) ;
```

где Screen - указатель на видеопамять, LeftHalf - ширина логической левой части изображения, а Bitmap - указатель на битовую карту изображения, загруженную в память. Этот процесс повторяется для каждой строки развертки изображения.

Каждый раз, когда вы увеличиваете значение LeftHalf на единицу, вы должны убедиться, что оно не превышает общей ширины изображения:

- Если LeftHalf больше ширины изображения, присвойте ей значение, равное единице;
- Если LeftHalf меньше единицы, присвойте ей значение, равное общей ширине изображения.

Запомните это, поскольку вы должны рисовать и левую и правую половину, причем ширина каждой логической части должна быть меньше общей ширины изображения.

Листинг 17.1 - это файл заголовка PARAL.H, в котором содержатся объявления различных констант, структур данных, а также прототипы функций, используемых в демонстрационной программе из Листинга 17.2 (PARAL.C).

Листинг 17.1. Файл заголовка демонстрационной программы циклического скроллинга (PARAL.H).

```
//
//Paral.h - данный заголовок определяет константы и структуры
//данных, используемые в демонстрационной программе
//            параллакса
#define KEYBOARD 0x09 //
//Коды клавиатуры для прерывания INT 9h
#define RIGHT_ARROW_PRESSED   77
#define RIGHT_ARROW_REL      205
#define LEFT_ARROW_PRESSED    75
#define LEFT_ARROW_REL       203
#define ESC_PRESSED          129
#define UP_ARROW_PRESSED      72
#define UP_ARROW_REL         200
#define DOWN_ARROW_PRESSED    80
#define DOWN_ARROW_REL       208
#define VIEW_WIDTH    320
#define VIEW_HEIGHT   150
#define MEMBLK        VIEW_WIDTH*VIEW HEIGHT
#define TRANSPARENT   0       // цветовые коды
#define TOTAL_SCROLL  320
enum (NORMAL, RLE},;
enum (FALSE,TRUE};
typedef struct
{
char manufacturer;   /* Всегда 0 */
char version;        /* Всегда 5 для 256-цветных файлов */
char encoding;       /* Всегда 1 */
char bits_per_pixel;
/* Должно быть равно 8 для 256-цветных файлов */
int  xmin, ymin;      /* Координаты левого верхнего угла */
int  xmax,ymax;      /* Высота и ширина образа */
int  hres;           /* Горизонтальное разрешение образа */
int  vres;           /* Вертикальное разрешение образа */
char palettel6[48];
/* палитра EGA; не используется для 256-цветных файлов */
char reserved;       /* зарезервировано */
char color planes;   /* цветовые планы */
int  bytes_per_line;
/* количество байт в каждой строке пикселей */
int  palette_type;
/* Должно быть равно 2 для цветовой палитры */
char filler[58];     /* Не используется */
} PcxHeader;
typedef struct
{
PcxHeader hdr;
char *bitmap;
```

```
char pal[768] ;
unsigned imagebytes,width,height;
} PcxFile;
#define PCX_MAX_SIZE 64000L enum {PCX_OK,PCX_NOMEM,PCX_TOOBIG,PCX_NOFILE};
#ifdef __cplusplus
extern "C" {
#endif
int ReadPcxFile(char *filename,PcxFile *pcx);
void _interrupt NewInt9(void) ;
void RestoreKeyboard(void);
void InitKeyboard(void);
void SetAllRgbPalette(char *pal);
void InitVideo (void);
void RestoreVideo(void);
int InitBitmaps(void); void FreeMem(void);
void DrawLayers(void);
void AnimLoop(void);
void Initialize(void);
void CleanUp (void) ;
void OpaqueBIt (char*, int, int, int) ;
void TransparentBit(char *,int,int,int) ;
#ifdef __cplusplus
} #endif
```

Программа из Листинга 17.2 (PARAL.C) демонстрирует повторяемое смещающееся изображение. Движущаяся картинка показывает облачное небо под солнцем. Хотя изображение и выглядит непрерывно меняющимся, но на самом деле оно неподвижно.

Наиболее важной частью программы является функция OpaqueBIt(). Она выводит левую и правую части изображения в буфер системной памяти, основываясь на значении LeftHalf. Когда построение закончено, содержимое буфера копируется на экран.

Запустив оттранслированную программу, используйте курсорные клавиши «влево» и «вправо» для изменения направления скроллинга. Для выхода из программы нажмите Esc. При этом она вычислит и покажет скорость анимации кадра. На машине с процессором 386SX/25 скорость выполнения составила около 35 кадров в секунду при размерах демонстрационного окна 320x100 Пикселей.

Листинг 17.2 Демонстрационная программа повторяемого смещения.

```
#include <stdio.h>
#include<stdlib.h>
#include<string.h>
#include <time.h>
#include<dos.h>
#include "paral.h"
char *MemBuf,            // указатель на буфер памяти
*BackGroundBmp,     // указатель на скрытую битовую карту
*VideoRam;          // указатель на память VGA
PcxFile pcx;             // структура PCX-файла
int volatile KeyScan;    // изменения клавиатурного обработчика
int frames=0,            // количество нарисованных кадров
PrevMode;            // сохраняет исходный видеорежим
int background;
void _interrupt (*OldInt9)(void); // указатель на клавиатурный
// обработчик BIOS
// Данная процедура загружает 256-цветный PCX-файл
int ReadPcxFile(char *filename,PcxFile *pcx)
{
long i;
int mode=NORMAL,nbytes;
char abyte,*p;
FILE *f;
f=fopen(filename,"rb") ;
if(f==NULL)
return PCX_NOFILE;
fread(&pcx->hdr,sizeof(PcxHeader),1,f) ;
pcx->width=1+pcx->hdr.xmax-pcx->hdr.xmin;
pcx->height=1+pcx->hdr.ymax-pcx->hdr.ymin;
pcx->imagebytes= (unsigned int) (pcx->width*pcx->height);
```

```
if(pcx->imagebytes > PCX_MAX_SIZE)
return PCX_TOOBIG;
pcx->bitmap= (char*)malloc (pcx->imagebytes);
if(pcx->bitmap == NULL)
return PCX_NOMEM;
p=pcx->bitmap;
for(i=0;i<pcx->imagebytes;i++)
{
if (mode == NORMAL)
{
abyte=fgetc(f);
if((unsigned char)abyte > 0xbf)
{
nbytes=abyte & 0x3f;
abyte=fgetc(f);
if(--nbytes > 0) mode=RLE;
}
}
else if(--nbytes == 0) mode=NORMAL;
*p++=abyte;
}
fseek(f,-768L,SEEK_END);      // извлечь палитру из PCX-файла
fread(pcx->pal,768,1,f) ;
p=pcx->pal;
for(i=0;i<768;i++)
*p++=*p >>2;
fclose(f) ;
return PCX_OK;                // успешное завершение
}
// Это новый обработчик прерывания 9h. Он позволяет осуществлять
// мягкий скроллинг. Если обработчик BIOS не будет запрещен,
// удержание клавиш управления курсором приведет к переполнению
// буфера клавиатуры и очень неприятному звуку из динамика.
void _interrupt NewInt9(void)
{
register char x;
KeyScan=inp(0x60);   // прочитать символ с клавиатуры
x=inp(0x61);      // сообщить клавиатуре, что символ обработан
outp(0x61,(x|0x80)) ;
outp(0x61,x);
outp (0x20,0x20} ;    // сообщить контроллеру
// прерываний о завершении прерывания
if(KeyScan == RIGHT ARROW REL ||  // проверить кнопки
KeyScan == LEFT_ARROW_REL)
KeyScan=0;
}
// Функция восстанавливает прежний обработчик прерываний клавиатуры
void RestoreKeyboard(void) {
_dos_setvect (KEYBOARD, OldInt9); // восстановить прежний вектор
}
// Эта функция сохраняет указатель вектора клавиатурного прерывания
// BIOS, а затем инсталлирует новый вектор прерывания, определенный
//в программе.
void InitKeyboard(void)
{
OldInt9=_dos_getvect(KEYBOARD); // сохранить вектор BIOS
_dos_setvect (KEYBOARD,NewInt9);// инсталлировать новый вектор
}
// Функция вызывает видео BIOS и заполняет все необходимые регистры
// для работы с палитрой, задаваемой массивом pal[]
void SetAllRgbPalette(char *pal)
{
struct SREGS s;
union REGS r;
segread(&s);                     // получить значение сегмента
s.es=FP_SEG((void far*)pal);    // ES указывает на pal
r.x.dx=FP_OFF((void far*)pal);  // получить смещение pal
```

```
r.x.ax=0xl012;                  // int l0h, функция 12h
// (установка регистров DAC)
r.x.bx=0;                      // первый регистр DAC
r.x.cx=256;                     // количество регистров DAC
int86x(0x10,&r,&r,&s);          // вызвать видео BIOS }
// Функция устанавливает видеорежим BIOS 13h
// это MCGA-совместимый режим 320x200x256 цветов
void InitVideo()
{
union REGS r ;
r.h.ah=0x0f;          // функция BIOS Ofh
int86(0x10,&r,&r);         // вызывать видео BIOS
PrevMode=r.h.al;           // сохранить текущий видеорежим
r.x.ax=0xl3;               // установить режим 13h
int86(0x10,&r,&r);         // вызвать видео BIOS
VideoRam=MK_FP(0xa000,0) ;  // создать указатель на видеопамять
}
// Функция восстанавливает изначальный видеорежим
void RestoreVideo() {
union REGS r;
r.x.ax=PrevMode;          // восстановить начальный видеорежиы
int86(0xl0,&r,&r);        // вызвать видео BIOS
}
// Функция загружает битовые карты
int InitBitmaps()
{
int r;
background=l;
r=ReadPcxFile("backgrnd.pcx",&pcx); // прочитать битовую карту
if(r != PCX_OK)        // выход при возникновении ошибки return FALSE;
BackGroundBmp=pcx.bitmap;    // сохранить указатель битовой
//  карты
SetAllRgbPalette(pcx.pal);   // установить палитру VGA
MemBuf=malloct(MEMBLK);       // выделить память под буфер
if(MemBuf == NULL)           // проверить на ошибки при
//  выделении памяти
return FALSE;
memset(MemBuf,0,MEMBLK);     // очистить
return TRUE;                 // Порядок!
}
// Функция освобождает выделенную программой память
void FreeMem()
{
free(MemBuf);
free(BackGroundBmp);
}
// функция рисует прокручиваемую битовую карту, не содержащую
// прозрачных пикселей; для скорости используется функция memcpyO;
// аргумент ScrollSplit задает столбец по которому битовая карта
// разбивается на две части
void OpaqueBlt(char *bmp,int StartY,int Height,int ScrollSplit)
{
char *dest;
int i;
dest=MemBuf+StartY*320; // вычисляем начальную позицию буфера
for(i=0;i<Height;i++)
{
// нарисовать левую половину битовой карты в правой половине буфера
menicpy(dest+ScrollSplit,bmp,VIEW_WIDTH-ScrollSplit) ;
// нарисовать правую половину битовой карты в левой половине буфера
memcpy(dest,bmp+VIEW_WIDTH-ScrollSplit,ScrollSplit);
bmp+=VIEW_WIDTH;
dest+=VIEW_WIDTH;
} } // конец функции
// Функция рисует смещающиеся слои
void DrawLayers()
{
```

```
OpaqueBlt(BackGroundBmp,0,100,background);
}
// Функция, обеспечивающая анимацию изображения.
// Наиболее критичная по времени выполнения.
// Для оптимизации как эту функцию, так и процедуры,
// которые она вызывает, рекомендуется переписать на ассемблере
// (В среднем это увеличивает производительность на 30%)
void AnimLoop()
{
while(KeyScan != ESC_PRESSED)    // цикл, пока не нажата ESC
{
switch(KeyScan)                  // обработать нажатую клавишу
{
case RIGHT_ARROW_PRESSED:      // нажата правая стрелка
background-=1;      // скроллировать фон на 2
// пикселя влево
if(background < 1)           // еще не конец образа?
background+=VIEW_WIDTH;    // ...тогда, можно смещать
// фон дальше
break;
case LEFT ARROW_PRESSED:       // нажата левая стрелка
background+=1;               // скроллировать фон на 2
// пикселя вправо
if(background > VIEW_WIDTH-1) // еще не конец образа
background-=VIEW_WIDTH;     // ...тогда можно смещать
// фон дальше
break;
default:                         // обработать все остальные
// клавиши break;
} DrawLayers();
memcpy(VideoRam,MemBuf,MEMBLK); // копировать MemBuf в
// VGA-память
frames++;
} }
// Функция осуществляет полную инициализацию
void Initialize()
{
InitVideo();            // установить режим 13h
InitKeyboard();         // установить собственный обработчик
// прерываний клавиатуры
if(!InitBitmaps())      // прочитать битовые образы
Cleanup();            // освободить память
printf("\nError loading bitmaps\n");
exit(l);
} }
// функция восстанавливает исходное состояние системы
void Cleanup()
{
RestoreVideo();       // восстановить VGA
RestoreKeyboard();    // восстановить вектор клавиатуры
FreeMem();            // освободить память
}
// Начало основной программы
int main()
{
clock_t begin, fini;
Initialize();
begin=clock();         // получить "тики" часов при старте
AnimLoop();            // начать анимацию изображения
fini=clock();          // получить "тики" часов в конце
Cleanup();             // освободить память
printf("Frames: %d\nfps: %gf\n", frames,
(float)CLK_TCK*frames/(fini-begin));
return 0;
}
```

Несколько смещающихся слоев

Вы теперь знаете, как перемещать изображение влево и вправо, а также осуществлять его циклический возврат к границам экрана. Следующим шагом попробуем смещать не один, а сразу несколько слоев изображения, причем двигать их будем с различными скоростями, что обеспечит иллюзию трехмерной графики.

Прежде всего, нужно уметь определять скорости движения различных слоев. Для простоты можно предположить, что скорость перемещения слоев линейно уменьшается с их «удалением» от наблюдателя. Это предположение удовлетворяет тому условию, что ближние слои движутся быстрее, чем дальние. Это справедливо и для предельного случая, когда бесконечно удаленный слой совершенно не перемещается. Так могут выглядеть звезды, облака или далекие горные цепи. Обратите внимание, что предположение о линейном изменений скорости слоев не вполне корректно, но, тем не менее, это является разумным компромиссом.

Не стоит пытаться вычислять точные скорости смещения (если вы сделаете это, то, скорее всего, получите число с плавающей запятой, а по известным причинам скорость смещения должна иметь целочисленное значение). В действительности демонстрационная программа параллакса из Листинга 17.3 использует простое правило для скоростей смещения: каждый следующий слой смещается вдвое медленнее, чем слой впереди него. Запомните: важно только то, чтобы относительное движение между различными слоями обеспечивало ощущение глубины пространства.

Изображение строится слой за слоем от заднего к переднему плану. В результате ближние слои перекрывают и прячут некоторые части более удаленных (этот метод, известный также как Алгоритм Художника, детально обсуждался в шестой главе, «Третье измерение»). Такое сокрытие дальних слоев ближними создает ощущение перспективы. Что же касается площади перекрытия, то она зависит от изображений. Функция OpaqueBIt() может быть использована также и для рисования перекрывающихся слоев. Плохо только, что при выводе изображения она стирает имеющуюся картинку. Это не очень практично для большинства типов декорации. Поэтому нам нужно научиться рисовать образы так, чтобы они закрывали собой уже выведенное изображение не полностью, а лишь по контуру и чтобы вся внешняя область оставалась без изменения.

«Прозрачные» пиксели

«Прозрачными» будем называть такие пиксели, которые при выводе на экран пропускаются и не перекрывают имеющееся изображение. Один из методов получения такого результата заключается в проверке значения цвета каждого пикселя перед тем, как он будет нарисован. Если цвет пикселя совпадает с «прозрачным», мы пропускаем данный пиксель и переходим к следующему. Такое дополнение к алгоритму ложится тяжелым бременем на нашу борьбу за скорость работы программы в процессе выполнения, а тем более — при выводе на экран. Ведь теперь мы не можем воспользоваться функцией memcpy() для вывода целой строки пикселей на экран, а должны применить цикл for() для изображения каждой точки отдельно.

Листинг 17.3 содержит новую функцию, называемую TransparentBlt(). Она заменит нам OpaqueBIt(). Разница между ними состоит только в том, что TransparentBlt() пропускает «прозрачные» пиксели (и это тоже тормозит работу программы).

Но как же TransparentBlt() отличает «прозрачные» пиксели от «непрозрачных»? Я решил, что любой пиксель со значением цвета, равным 0 (обычно, это черный) будет «прозрачным», но вы можете назначить для этого другой цвет. Функция пропускает любой пиксель, у которого значение цвета равно объявленной константе TRANSPARENT. Программа из Листинга 17.3 (PARAL1.C) является демонстрацией смещения двух повторяющихся слоев. Дальний слой сплошной, в то время как ближний включает в себя «прозрачные» пиксели. Для вывода изображений используются функции OpaqueBIt() и TransparentBit() соответственно. Несмотря на то, что у нас имеется всего два движущихся слоя, эффект получается довольно реалистичным. Как и в программе из Листинга 17.2, курсорные клавиши «влево» и «вправо» перемещают изображение по горизонтали, а для завершения программы нужно нажать Esc.

Обратите внимание, что скорость смены кадров в этой программе значительно ниже, чем в предыдущей. Это происходит из-за использования функции для работы с «прозрачными» пикселями. На компьютере с процессором 386SX/25 я получил примерно 10 кадров в секунду. В принципе, это не так уж и плохо для программы, написанной полностью на Си.

Листинг 17.3. Простой двойной параллакс (PARAL1.C).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <dos.h>
#include "paral.h"
char *MemBuf,            // указатель на дублирующий буфер
*BackGroundBmp,     // указатель на битовую карту фона
*ForeGroundBmp,     // указатель на битовую карту
```

```
// ближнего плана
*VideoRam;          // указатель на видеобуфер
PcxFile pcx;             // структура данных
// для чтения PCX-файла
int volatile KeyScan;    // заполняется обработчиком
// прерывания клавиатуры
int frames=0,            // количество нарисованных кадров
PrevMode;            // исходный видеорежим
int background, // позиция прокрутки фона
foreground, //позиция прокрутки битовой карты
// ближнего плана position;    // общее расстояние прокрутки
void _interrupt (*OldInt9)(void); // указатель на обработчик
// прерывания клавиатуры BIOS
// Функция загружает 256 - цветный PCX-файл
int ReadPcxFile(char *filename,PcxFile *pcx)
{
long i;
int mode=NORMAL,nbytes;
char abyte,*p;
FILE *f;
f=fopen(filename,"rb");
if(f==NULL)
return FCX_NOFILE;
fread(&pcx->hdr,sizeof(PcxHeader),1, f);
pcx_width=1+pcx->hdr.xmax-pcx->hdr.xmin;
pcx->height=1+pcx->hdr.ymax-pcx->hdr.ymin;
pcx->imagebytes=(unsigned int) (pcx->width*pcx->height);
if(pcx->imagebytes > PCX_MAX_SIZE) return PCX_TOOBIG;
pcx->bitmap= (char*)malloc (pcx->imagebytes);
if(pcx->bitmap == NULL) return PCX_NOMEM;
p=pcx->bitmap;
for(i=0;i<pcx->imagebytes;i++)
{
if(mode == NORMAL)
{
abyte=fgetc(f);
if((unsigned char)abyte > 0xbf)
{ nbytes=abyte & 0x3f;
abyte=fgetc(f);
if(--nbytes > 0)
mode=RLE;
}
}
else if(-—nbytes == 0) mode=NORMAL;
*p++=abyte;
}
fseek(f,-768L,SEEK_END);      // получить палитру,из PCX-файла
fread(pcx->pal,768,1,f);
p=pcx->pal;
for(i=0;i<768;i++) // битовый сдвиг цветов в палитре
*p++=*p >>2;
fclose(f) ;
return PCX_OK;
}
// Новый обработчик прерывания клавиатуры для программы прокрутки
// Он используется для интерактивной прокрутки изображения.
// ЕСЛИ стандартный обработчик прерывания 9h не будет заблокирован
// длительное нажатие на клавиши управления курсором приведет
// к переполнению буфера клавиатуры и появлению крайне неприятного
// звука из динамика.
void _interrupt Newlnt9(void)
{
register char x;
KeyScan=inp(0x60);// прочитать код клавиши
x=inp(0x61);      // сообщить клавиатуре, что символ обработан
outp(0x61, (x|0x80));
outp(0x61,x);
```

```
outp(0x20,0x20);  // сообщить о завершении прерывания
if(KeyScan == RIGHT_ARROW_REL ||// проверка кода клавиши
KeyScan == LEFT_ARROW_REL) KeyScan=0;
}
// Функция восстанавливает исходный обработчик прерываний клавиатуры
void RestoreKeyboard(void)
{
_dos_setvect(KEYBOARD,OldInt9);   // восстанавливаем
// обработчик BIOS
}
// Эта функция сохраняет прежнее значение вектора прерывания // клавиатуры и
устанавливает новый обработчик нашей программы.
void InitKeyboard(void)
{
OldInt9= _dos_getvect(KEYBOARD);   // сохраняем адрес
//  обработчика BIOS
_dos_setvect(KEYBOARD,NewInt9);   // устанавливаем новый
//  обработчик прерывания 9h
}
// Эта функция использует функции BIOS для установки в регистрах
// видеоконтроллера значений, необходимых для работы с цветами,
// определяемыми массивом pal[]
void SetAllRgbPalette(char *pal)
{
struct SREGS s;
union REGS r;
segread(&s);  // читаем текущее значение сегментных регистров
s.es=FP_SEG((void far*)pal);  // в ES загружаем сегмент pa1[]
r.x.dx=FP OFF((void far*}pal);// в DX загружаем смещение pal[]
r.x.ax=0xl012;         // готовимся к.вызову подфункции // 12h функции BIOS 10h
r.x.bx=0;            /;/ номер начального регистра палитры
r.x.cx=256;           // номер последнего изменяемого регистра
int86x(0xl0,&r,&r,&s);// вызов видео BIOS
}
// Функция устанавливает режим 13h
// Это MCGA-совместимыЙ режим 320x200x256 цветов
void InitVideo()
{
union REGS r;
r.h.ah=0x0f;       // функция Ofh - установка видеорежима
int86(0xl0,&r,&r); // вызов видео BIOS
PrevMode=r.h.al;   // сохраняем старое значение режима
r.x.ax=0xl3;       // устанавливаем режим 13h
int86(0x10,&r,sr); // вызов видео BIOS
VideoRam=MK_FP(0xa000,0); // создаем указатель на видеопамять
}
//Эта функция восстанавливает исходный видеорежим
void RestoreVideo()
{
union REGS r;
r.x,ax=PrevMode;   //исходный видеорежим
int86(0x10,&r,&r); // вызов видео BIOS
}
// Функция загрузки битовых карт слоев
int InitBitmaps()
{
int r;
// начальное положение линии деления
background=foreground=1;
// читаем битовую карту фона
r=ReadPcxFile("backgrnd.pcx",&pcx);
// проверка на ошибки чтения if(r != PCX_OK)
return FALSE;
// запоминаем указатель на битовую карту
BackGroundBmp=pcx.bitmap;
// устанавливаем палитру
SetAllRgbPalette(pcx.pal) ;
```

```
// читаем битовую карту переднего слоя
r=ReadPcxFile("foregrnd.pcx",&pcx);
// проверка на ошибки чтения
if (r != PCX_OK) return FALSE;
//запоминаем указатель на битовую карту
ForeGroundBmp=pcx.bitmap;
// создаем буфер в памяти
MemBuf=malloc(MEMBLK);
// проверка на ошибки распределения памяти
if(MemBuf == NULL) return FALSE;
memset(MemBuf,0,MEMBLK); // очистка буфера
return TRUE;
// все в порядке!
}
// функция освобождает выделенную память
void FreeMem()
(
free(MemBuf);
free(BackGroundBmp);
free(ForeGroundBmp);
}
// Функция рисует слои параллакса.
// Порядок отрисовки определяется координатой слоя по оси Z.
void DrawLayers()
{
OpaqueBlt(BackGroundBmp,0,100,background);
TransparentBlt(ForeGroundBmp,50,100,foreground);
}
// Эта функция осуществляет анимацию. Учтите, что это наиболее
// критичная по времени часть программы. Для оптимизации отрисовки
// как сама функция, так и те функции, которые она вызывает,
// следует переписать на ассемблере. Как правило, это увеличивает
// быстродействие на 100 процентов.
void AnimLoop()
{
while(KeyScan != ESC_PRESSED)    // пока не нажата клавиша ESC
(
switch(KeyScan)     // определяем, какая клавиша была нажата
{
case RIGHT_ARROW_PRESSED:      //нажата "стрелка вправо"
position--;                   // изменяем позицию
if(position < 0)              // останавливаем прокрутку,
//  если дошли до конца
{
position=0;
break;
} backgrpund –=1;      // прокручиваем фон влево на 2 пикселя
if(background < 1)         // дошли до конца?
background+=VIEW_WIDTH; // ...если да - возврат к началу
foreground-=2;             // прокручиваем верхний
// слой влево на 4 пикселя
if(foreground < 1)         // дошли до конца?
foreground+=VIEW_WIDTH; // ...если да - возврат к началу
break;
case LEFT_ARROW_PRESSED:      // нажата "стрелка влево"
position++;          // изменяем текущую позицию прокрутки
if(position > TOTAL_SCROLL) // останавливаем прокрутку,
// если дошли до конца
{
position=TOTAL_SCROLL;
break;
}
background+=l;     // прокручиваем фон вправо на 2 пикселя
if(background > VIEW_WIDTH-1) // дошли до конца?
background-=VIEW_WIDTH; // ...если да - возврат к началу
foreground+=2;             // прокручиваем верхний слой
// вправо на 4 пикселя
```

```
if(foreground > VIEW_WIDTH-1) // дошли до конца?
foreground-=VIEW_WIDTH; // ...если да - возврат к началу
break;
default:                    // игнорируем остальные клавиши
break;
}
DrawLayers();                // рисуем слои в буфере в
// оперативной памяти
memcpy(VideoRam,MemBuf,MEMBLK); // копируем буфер в
// видеопамять
frames++;                    // увеличиваем счетчик кадров
) }
//эта функция осуществляет необходимую инициализацию
void Initialize()
{
position=0;
InitVideo();            // устанавливаем видеорежим 13h
InitKeyboard();         // устанавливаем наш обработчик
// прерывания клавиатуры
if(!InitBitmaps())      // загружаем битовые карты
{
CleanUp();        //освобождаем память
printf("\nError loading bitmaps\n");
exit(1);
} }
// функция выполняет всю необходимую очистку
void Cleanup()
{
RestoreVideo();       // восстанавливаем исходный видеорежим
RestoreKeyboard();    // восстанавливаем обработчик
// прерывания клавиатуры BIOS
FreeMem();            // освобождаем всю выделенную память
}
// Это начало программы. Функция вызывает процедуры инициализации.
// Затем читает текущее значение системного таймера и запускает
// анимацию. Потом вновь читается значение системного таймера.
// Разница между исходным и конечным значениями таймера
// используется для вычисления скорости анимации.
int main()
{
clock_t begin,fini;
Initialize(};          // проводим инициализацию
begin=clock();         // получаем исходное значение таймера
AnimLoop();            // выполняем анимацию
fini=clock();          // получаем значение таймера
CleanUp();             // восстанавливаем измененные параметры
printf("Frames: %d\nfps: %f\n",frames,
(float)CLK_TCK*frames/(fini-begin));
return 0;
}
```

Оптимизированные версии OpaqueBlt() и TransparentBlt()

Листинг 17.4 содержит оптимизированные ассемблерные версии подпрограмм OpaqueBlt() и TransparentBlt(), которые имеются и на дискете (BLIT.ASM). Эти подпрограммы можно использовать вместо соответствующих функций на Си, что увеличит быстродействие программы примерно на 30 процентов.

Демонстрационная программа PARAL использует ассемблерные версии этих подпрограмм. Они были написаны с тем расчетом, чтобы могли работать с процессором 286. Поскольку они заполняют буфер системной памяти, их можно оптимизировать и дальше, применив 32-битовые команды перемещения банных.

Листинг 17.4. Оптимизированная подпрограмма (BLIT.ASM).

```
ideal
model compact,c
p286
dataseg
VIEW_WIDTH equ 320
VIEW_HEIGHT equ 100
```

```
TRANSPARENT equ 0
global MemBuf:dword
.codeseg
public OpaqueBIt public TransparentBIt
;
;Эта процедура копирует битовую карту в MemBuf. Кроме этого, она
; может скроллировать левую и правую части битового образа
; в зависимости от значения ScrollSplit ;
proc OpaqueBlt
ARG Bitmap:dword,StartY:word,Height:word,ScrollSplit:word
USES si,di
les   di,[MemBuf]      ; получить указатель на буфер памяти
mov   ax,[StartY]      ;получить начальную Y-координату
mov   bx,ax            ;скопировать
sal   ax,6             ; умножить на 64
sal   bx,8             ; умножить на 256
add   ax,bx            ; результат равен умножению на 320
add   di,ax            ; вычислить смещение внутри MemBuf
mov   bx,[Height]      ; получить высоту битовой карты
mov   dx,[ScrollSplit] ; получить длину правой половины
push ds                ; сохранить сегмент данных
lds   si,[Bitmap]     ; получить полный указатель на битовую карту
mov   ax,VIEW_WIDTH    ; получить ширину экрана
sub   ax,dx            ; вычислить длину левой половины
cld        ; обрабатывать данные от младших адресов к старшим
@@loop01: add   di,dx            ; вычислить начальную позицию
mov   cx,ax            ; получить длину левой половины
shr   cx,1 ; разделить на 2 (поскольку выводим по 2 байта)   :
rep   movsw            ; нарисовать правую половину карты
jnc   short @@skip01   ; пропустить, если выполнено
movsb                  ; дорисовать последний пиксель
@@skip01:
sub   di,VIEW_WIDTH    ; получить ширину выводимого окна
mov   cx,dx            ; получить длину правой половины
shr  cx,1             ; разделить на 2
rep   movsw            ; нарисовать правую половину карты
jnc   short @@skip02   ; пропустить, если выполнено
movsb                  ; перейти к последнему пикселю
@@skip02:
add   di,ax         ; получить ширину следующей строки вывода
dec   bx               ; уменьшить счетчик столбцов
jnz   short @@loop01   ; повторить
pop   ds               ; восстановить сегмент данных
ret                    ; завершить
endp OpaqueBlt
;
; Эта процедура копирует битовую карту в MemBuf. Функция не рисует
; пиксель, если его цвет равен "прозрачному"
;
proc TransparentBIt
ARG Bitmap:dword,StartY:word,Height:word,ScrollSplit:word USES si,di
les   di,[MemBuf]      ; получить указатель на память
mov   ax,[StartY]      ; получить начальную Y-координату
mov   ex, ax            ; получить копию
sal   ax,6             ; умножить на 64
sal   ex,8             ; умножить 256
add   ax, ex          ; результат эквивалентен умножению на 320
add   di,ax            ; прибавить смещение к MemBuf
mov   dx,[ScrollSplit] ; получить ширину левой половины
                        ;  изображения
mov   bx,VIEW_WIDTH    ; общая ширина изображений
sub   bx,dx            ; вычислить ширину правой половины
push  ds               ; сохрани,ть сегмент , данных
lds   si,[Bitmap] ; получить полный указатель на битовый образ
@@loop01;
add   di,dx            ; вычислить начальную позицию экрана
mov   cx,bx            ; получить ширину правой половины ;
```

```
;Нарисовать правую половину ;
@@loop02;
mov   al,[si]            ; получить пиксель карты
inc   si                 ; следующий пиксель
cmp   al,TRANSPARENT     ; "прозрачный"?
je    short @@skip01     ; ...пропустить, если да
mov   [es:di],al         ; иначе, нарисовать его
@@skip01:
inc   di                 ; указать на следующий байт MemBuf
dec   ex                 ; уменьшить счетчик
jnz   short @@loop02
sub   di,VIEW_WIDTH       ; получить ширину выводимого окна
mov   cx,dx               ; получить ширину правой половины
; Нарисовать левую половину
@@loop03:
mov   ai,[si]            ; получить пиксель карты
inc   si                 ; следующий пиксель
cmp   al,TRANSPARENT     ; "прозрачный"?
je    short @@skip02       ;...пропустить, если да
mov   [es:di],al           ;иначе, нарисовать его
@@skip02;
inc   di                   ;перейти к следующему байту видеобуфера
dec   ex                   ;уменьшить счетчик
jnz   short @@loop03       ;повторить цикл, пока не сделаем все
add   di,bx                ;добавить значение ширины правой
части изображения
dec   [Height]             ;уменьшить счетчик колонок
jnz   short @@loop01       ;повторить еще раз
pop   ds                   ;восстановить сегмент данных
ret                       ; закончить
endp TransparentBlt
end
```

Смещение мозаичных изображений

Одной из проблем, возникающей при выводе повторяющегося смещаемого изображения, является большой объем памяти, требуемый для размещения битовых карт. Практически, размер никакого отдельного изображения в D0b не может превышать 64K. Этого достаточно для изображений площадью 320x200 или 640x100 пикселей (как я уже говорил, в режиме 13h каждый пиксель занимает один байт). Но даже если бы вам и представилась возможность иметь изображения больших размеров, вы все равно очень скоро исчерпали бы память, поскольку в вашем распоряжении имеется максимум 640K.

Не волнуйтесь: мы не будем здесь углубляться в изучение 16-битного кода, расширенной памяти или сегментации. Скажу только, что ограничения использования памяти неизменно сопутствуют старому доброму реальному режиму работы микропроцессора. Когда-нибудь это может измениться, но сейчас мы вынуждены примириться с таким положением вещей.

Существует довольно умное решение проблемы недостатка памяти — это применение мозаичных изображений. Вместо использования цельного образа, вы создаете виртуальную битовую карту, составленную из множества меньших картинок, как. из кирпичиков. Эти «кирпичики» могут быть рассмотрены как блоки здания, из которых составляется большее изображение. В основе этого технического приема лежит составление специальной справочной таблицы. Обычно она представляет собой массив, содержащий данные о расположении маленьких частей внутри большого виртуального изображения. Справочная таблица не требует много памяти (приблизительно один или два байта на элемент) и поэтому может иметь практически любую протяженность.

Представьте себе смещающееся виртуальное изображение, состоящее из 5400x12 «кирпичиков», каждый из которых имеет размер всего 16x16 пикселей. Это означает, что площадь виртуального изображения составит 86400x192 пикселя, что намного больше, максимальных допустимых размеров любого отдельного изображения.

Единственным ограничением этого метода является то обстоятельство, что размер цельного образа полностью зависит от размеров отдельных «кирпичиков». Каждый из них должен быть небольшим и достаточно типовым, чтобы имелась возможность использования его в различных местах изображения. Тем не менее, «кирпичики» должны быть достаточно интересными, чтобы из них можно было сформировать привлекательное изображение.

По практическим соображениям кирпичики должны иметь ширину, равную степени числа 2. То есть их размер по горизонтали должен составлять 2, 4, 16 и так далее вплоть до 320 пикселей в режиме 13h. Важность этих ограничений вы поймете позже.

Одни из «кирпичиков», составляющих изображение, могут включать в себя «прозрачные»

пиксели, а другие — нет. Последние наиболее пригодны для изображения дальних слоев, а также ближних планов, у которых отсутствуют «прозрачные» области, в то время как «кирпичики», содержащие «прозрачные» пиксели, служат для рисования частей изображения, имеющих пустоты. Но на самом деле «прозрачными» могут быть не только отдельные пиксели, но и целые "кирпичики", которые при выводе вообще пропускаются. Они могут располагаться в тех участках рисунка, где полностью отсутствует какое-либо изображение.

Помните, что вывод мозаичных слоев, включающих в себя "прозрачные" области, выполняется медленнее, потому что добавочные накладные расходы требуются для проверки существования «прозрачных» пикселей. В этой главея не показываю программу вывода «прозрачных» мозаичных слоев, но вы можете сделать это сами, применив технику вывода «прозрачных» пикселей. В своих собственных программах вы можете выбирать любые комбинации цельных и мозаичных слоев, «сплошных» или включающих в себя «прозрачные» пиксели, только не забывайте о различиях в способах их вывода.

При создании мозаичного изображения нужно учитывать тот факт, что на экране никогда не присутствует более двух «кирпичиков», выведенных не полностью. Рисунок 17.2 поясняет это свойство.

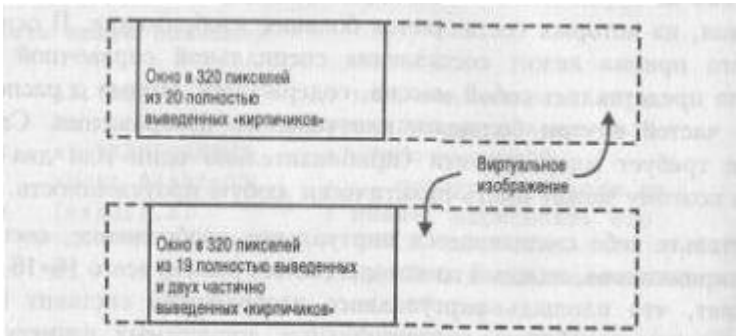


Рис. 17.2. *Свойства мозаичной матрицы изображений.*

Это свойство несколько упрощает построение мозаичного изображения. Таким образом, этот процесс состоит из трех шагов:

1. Рисование первого (возможно, не полностью выведенного) «кирпичика»;
2. Рисование нескольких (полностью выведенных) «кирпичиков»;
3. Рисование последнего (возможно, не полностью выведенного) «кирпичика» .

Также нам необходимо иметь в программе целочисленный счетчик, который будет указывать местоположение видимого экрана внутри виртуального изображения.

Программа из Листинга 17.6 (TILES.C) демонстрирует моделирование мозаичных слоев. Самые ближние слои состоят из нескольких «кирпичиков». Определение виртуального изображения сохранено в файле TILEMAP.DAT, который представляет собой обычный ASCII-файл и обрабатывается во время инициализации. Цифры в файле представляют собой закодированные имена PCX-файлов. Обратите внимание, что код 0 зарезервирован для «прозрачного кирпичика». Рисунок 17.3 показывает небольшой пример мозаичного изображения.

Важным отличием между этой программой и демонстрационной программой двойного параллакса в Листинге 17.3 является добавление функции DrawTile().

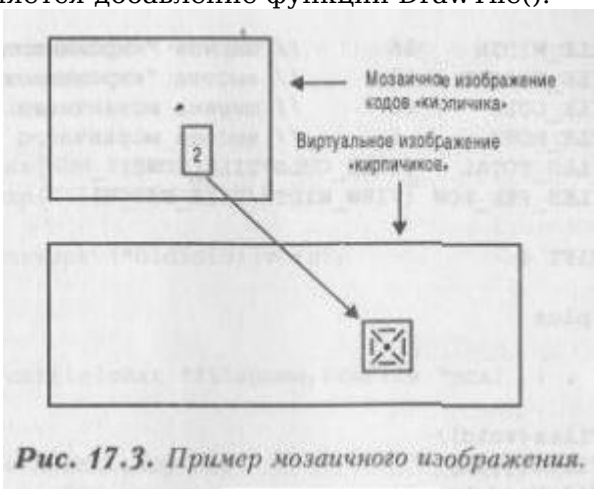


Рис. 17.3. *Пример мозаичного изображения.*

Эта подпрограмма изображает «кирпичик» в указанном месте экрана. Два аргумента offset и width определяют соответственно начальный столбец и ширину для вывода не полностью помещающихся на экране «кирпичиков».

Для частично выведенных кирпичиков:

- offset -первый столбец, в котором будет нарисован «кирпичик»;
- width - некоторое значение меньше ширины «кирпичика».

Для полностью выведенных «кирпичиков»:

- offset - 0;
- width - определяет ширину «кирпичика».

Программа из Листинга 17.6 также использует курсорные клавиши для управления движением и клавишу Esc для выхода. В демонстрационной программе на переднем плане появляется стена дома, составленная из отдельных «кирпичиков», а за ней видны хорошо известная горная гряда и линия неба на самом дальнем плане. Выполняется эта программа немного медленнее из-за использования в ней функции вывода мозаичного изображения, но только на пару кадров в секунду. Листинг 17.5 содержит файл заголовка для программы Построения мозаичного изображения, представленной в Листинге 17.6. В заголовке определены константы и прототипы функций для демонстрационной Программы.

Листинг 17.5. Заголовок мозаичного смещения (TILES.H).

```
// Этот файл содержит определения, используемые программой
// прокрутки мозаичных изображений
#define NUM_TILES     17      // количество файлов,
// содержащих "кирпичики"
#define TILE_WIDTH    16      //ширина "кирпичиков"
#define TILE_HEIGHT   16      // высота "кирпичиков"
#define TILE_COLS     40      //ширина мозаичного изображения
#define TILE_ROWS      6      // высота мозаичного изображения
#define TILES_TOTAL   (TILE_COLS*TILE_ROWS)
#define TILES_PER_ROW (VIEW_WIDTH/TILE_WIDTH)
#define SHIFT 4
ftifdef _cplusplus extern "C"
{
#endif
void ReadTiles(void);
void FreeTiles(void);
void ReadTileMap(char *);
void DrawTile(char *,int,int,int,int);
void DrawTiles(int,int);
#ifdef __cplusplus
}
#endif
```

Поскольку программа из Листинга 17.6 практически повторяет 17.3, она приводится без комментариев.

Листинг 17.6. Демонстрационная программа мозаичного смещающегося слоя (TILES.C).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <dos.h>
#include "paral.h"
#include "tiles.h"
char *MemBuf,
*BackGroundBmp,
*ForeGroundBnip,
*VideoRam;
PcxFile pcx;
int volatile KeyScan;
int frames=0,
PrevMode;
int background,
foreground, position;
char *tiles[NUM_TILES+l];
int tilemap[TILES_TOTAL] ;
void interrupt (*OldInt9)(void);
//
//
int ReadPcxFile(char *filename,PcxFile *pcx)
{
long i;
int mode=NORMAL,nbytes;
char abyte,*p;
FILE *f;
```

```
f=fopen(filename,"rb");
if(f==NULL)
return PCX_NOFILE;
fread(&pcx->hdr,sizeof(PcxHeader),l,f);
pcx->width=1+pcx->hdr.xmax-pcx->hdr.xmin;
pcx->height=1+pcx->hdr.ymax-pcx->hdr.ymin;
pcx->imagebytes= (unsigned int) (pcx->width*pcx->height) ;
if(pcx->imagebytes > PCX_MAX_SIZE)
return PCX_TOOBIG;
pcx->bitmap= (char*)malloc(pcx->imagebytes);
if(pcx->bitmap == NULL)
return PCX_NOMEM;
p=pcx->bitmap;
for(i=0;i<pcx->imagebytes;i++)
{
if(mode == NORMAL)
{
abyte=fgetc(f);
if((unsigned char)abyte > Oxbf)
{ nbytes=abyte & 0x3f;
abyte=fgetc(f);
if(-—nbytes > 0) mode=RLE ;
}
}
else if(--nbytes ==0)
mode=NORMAL;
*p++=abyte;
}
fseek(f,-768L,SEEK_END);
fread(pcx->pal,768,1, f) ;
p=pcx->pal;
for(i=0;i<768;i++) *p++=*p>>2;
fclose(f) ;
return PCX_OK;
}
//
void _interrupt NewInt9(void) {
register char x;
KeyScan=inp(Ox60);
x=inp(0x61) ;
outp(0x61,(x|0x80));
outp(0x61,x) ;
outp(0x20,0x20);
if(KeyScan == RIGHT__ARROW__REL ||
KeyScan == LEFT__ARROW_REL)
KeyScan=0;
}
//
void RestoreKeyboard(void) {
_dos_setvect(KEYBOARD,OldInt9);
}
//
void InitKeyboard(void)
{
Oldlnt9=_dos_getvect(KEYBOARD) ;
_dos_setvect(KEYBOARD,Newlnt9);
}
//
void SetAllRgbPalettefchar *pal)
{
struct SREGS s;
union REGS r;
segread(&s) ;
s.es=FP_SEG((void far*)pal);
r.x.dx=FP_OFF((void far*)pal);
r.x.ax=0xl012;
r.x.bx=0;
```

```
r.x.cx=256;
int86x(0xl0,&r,&r,&s) ;
}
//
void InitVideo()
{
union REGS r;
r.h.ah=0x0f;
int86(0xl0,&r,&r); PrevMode=r.h.al;
r.x.ax=0xl3;
int86(0xl0,&r,&r);
VideoRam=MK_FP(0xa000,0);
}
//
void RestoreVideo()
{
union REGS r;
r.x.ax=PrevMode;
int86(0xl0,&r,&r) ;
}
//
int InitBitmaps()
{
int r;
background=foreground=l;
r=ReadPcxFile("backgrnd.pcx",&pcx) ;
if(r != PCX_OK) return FALSE;
BackGroundBnip=pcx.bitmap ;
SetAllRgbPalette(pcx.pal); ,
r=ReadPcxFile("foregrnd.pcx",&pcx);
if(r != PCX_OK) return FALSE;
ForeGroundBmp=pcx.bitmap;
MemBuf=malloc(MEMBLK) ;
if(MemBuf == NULL) return FALSE;
memset(MemBuf,0, MEMBLK) ;
return TRUE;
) //
void FreeMem()
{
free(MemBuf);
free(BackGroundBmp) ;
free(ForeGroundBmp) ;
FreeTiles(};
}
//
void DrawLayers()
{ OpaqueBlt(BackGroundBmp,0,100,background) ;
TransparentBIt(ForeGroundBmp,50,100,foreground) ;
DrawTiles(position,54) ;
}
//
void AnimLoop() {
while(KeyScan != ESC_PRESSED)
{
switch(KeyScan)
{ case RIGHT_ARROW_PRESSED:
position+=4;
if(position > TOTAL_SCROLL) {
position=TOTAL_SCROLL;
break;
}
background-=1;
if(background < 1)
background+=VIEW_WIDTH;
foreground-=2; if(foreground < 1)
foreground+=VIEW_WIDTH;
break;
```

```
case LEFT_ARROW_PRESSED:
position-=4;
if(position < 0) {
position=0;
break;
}
background+=1;
if(background > VIEW_WIDTH-1) background-=VIEW_WIDTH;
foreground+=2 ;
if (foreground > VIEW_WIDTH-1) foreground-=VIEW_WIDTH;
break;
default:
break;
} DrawLayers();
memcpy(VideoRam,MemBuf,MEMBLK) ;
frames++;
} }
//
void Initialize()
{
position=0;
InitVideo(} ;
InitKeyboard();
if(!InitBitmaps())
{
Cleanup();
printf("\nError loading bitmaps\n");
exit(l);
} ReadTileMap("tilemap.dat");
ReadTiles();
}
//               void Cleanup() {
RestoreVideo() ;
RestoreKeyboard();
FreeMem();
}
void ReadTiles(void)
{
PcxFile pcx;
char buf[80];
int i,result;
tiles[0]=NULL;
for(i=l;i<=NUM_TILES;i++)
{
sprintf(buf,"t%d.pcx",i);
result=ReadPcxFile(buf,&pcx);
if(result != PCX_OK) ( printf("\nerror reading file: %s\n",buf);
exit(1);
} tiles[i]=pcx.bitmap;
} }
void FreeTiles() { int i;
for(i=0;i<NUM_TILES;i++) free(tiles[i]) ;
}
void ReadTileMap(char *filename)
{
int i;
FILE *f;
f=fopen(filename,"rt") ;
for (i=0; i<TILES__TOTAL; i-H-) {
fscanf(f,"%d",&(tilemap[i])) ;
}
fclose(f);
}
//
void DrawTile(char *bmp,int x,int y,int offset, int width)
{
char *dest;
```

```
int i;
if(bmp == NULL) return;
dest=MemBuf+y*VIEW_WIDTH+x;
bmp+=offset;
for(i=0;i<TILE_HEIGHT;i++) {
memcpy(dest,bmp,width);
dest+=VIEW_WIDTH;
bmp+=TILE_WIDTH;
} }
//
void DrawTiles(int VirtualX,int Starty)
{
int i,x,index,offset,row,limit;
index=VirtualX>>SHIFT;
offset=VirtualX - (index<<SHIFT) ;
limit=TILES_PER_ROW;
if(offset==0)
limit--;
for(row=Starty;row<Starty+TILE_HEIGHT*TILE_ROWS;
row+=TILE_HEIGHT) {
x=TILE_WIDTH-of£set;
DrawTile(tiles[tilemap[index]],0,row,offset,
TILE_WIDTH-offset);
for(i=index+l;i<index+limit;i++)
{
DrawTile(tiles [tilemap [i]], x, row, 0, TILE_WIDTH) ;
x+=TILE_WIDTH;
} DrawTile(tiles [tilemap[i] ] ,x, row, 0,offset);
index+=TILE_COLS;
}
}
//
int main() { clock_t begin,fini;
Initialize() ;
begin=clock();
AnimLoop() ;
fini=clock() ;
Cleanup() ;
printf("Frames: %d\nfps: %f\n",frames,
(float)CLK_TCK*frames/(fini-begin)) ;
return 0;
)
```

Устранение эффекта сдвига кадра

На медленных машинах или на машинах с медленными видеокартами можно заметить некий сдвиг изображения, как будто оно копируется на экран. Из-за эффекта сдвига изображение выглядит как бы разорванным. Этот интересный но нежелательный эффект появляется оттого, что адаптер сканирует видеобуфер и рисует изображение на дисплее примерно 60 раз в секунду. Этот процесс называется регенерацией экрана. Если программа в момент начала регенерации дисплея находится в процессе рисования кадра, вы заметите эффект сдвига изображения.

К счастью, существуют методы проверки статуса регенерации экрана. На VGA-карте есть регистр, сообщающий, регенерируется ли экран в настоящее время- Все, что требуется для устранения эффекта сдвига кадра, это подождать, пока регенерация экрана завершится. Затем можно начать рисовать изображение.

В Листинге 17.7 содержится фрагмент программы, ожидающей завершения цикла регенерации экрана. Это дает вам примерно 1/60 секунды, чтобы нарисовать следующий кадр. Данный фрагмент можно поместить непосредственно перед функцией, перемещающей кадр из системной памяти в видеобуфер. Выполняйте такую проверку каждый раз перед копированием буфера на экран. Только на очень быстрых машинах или при использовании небольшого окна вывода, одной шестидесятой секунды будет достаточно для изображения нескольких планов и их копирования на экран. Это главный недостаток режима 13h. Единственная альтернатива проверке на регенерацию экрана — это использование видеорежимов, поддерживающих несколько видеостраниц, и переключение между ними.

Листинг 17.7. Проверка вертикальной трассировки.

```
asm mov dx,0x3da
```

```
NoRetrace:
asm in al,dx
asm and al,8
asm jz NoRetrace    // ждать, пока трассировка завершится
Retrace:
asm in al,dx
asm and al,8
asm jnz Retrace     // ждать начала трассировки
```

Программа в этой главе не выполняет проверку вертикальной трассировки. Это было сделано для того, чтобы свести к минимуму использование ассемблера. Настоятельно рекомендую использовать этот фрагмент во всех программах вывода графики.

PCX-графика

Для удобства все картинки, использованные в этой главе, сохранены на диске как 256-цветные PCX-файлы. Формат PCX был выбран главным образом потому, что его легко читать и данные изображения в нем содержатся в сжатом виде. Немаловажно и то, что формат PCX поддерживается большинством графических редакторов (для получения более полной информации по PCX-файлам смотрите пятую главу, "Секреты VGA-карт").

Примечания по выполнении

Быстродействие нашей программы смещения страдает отчасти потому, что она часто рисует один и тот же пиксель более одного раза. Отлаживая программу в профайлере, вы можете заметить, что большая часть времени тратится на выполнение операции копирования системного буфера в видеопамять. Но на самом деле в большинстве случаев затраты времени на копирование готового изображения в видеобуфер оказываются ничтожными по сравнению с затратами на многократную перерисовку одних и тех же пикселей.

Это происходит потому, что время доступа к системной памяти значительно выше, чем к области видеобуфера. Однако, это не всегда справедливо. Некоторые адаптеры, такие как LOCAL BUS, работают значительно быстрее.

Существует несколько путей повышения производительности программ. Где только возможно, вы должны избегать вызовов функций и вместо этого использовать макросы. Команды, в которых отсутствуют вызовы функций, не содержат накладных расходов. Другой удар по быстродействию — это обилие аргументов, передаваемых функциям. Параметры передаются через стек, и каждая операция обмена со стеком отнимает драгоценное время. Несколько удачно подобранных глобальных переменных могут значительно увеличить скорость выполнения некоторых критических циклов.

Всякий раз, когда есть возможность, используйте функции для обработки блоков памяти. Они максимально оптимизированы и предназначены как раз для быстрого перемещения данных. Наиболее удобно применять их для рисования строк пикселей.

Если вам никак не обойтись без цикла FOR, попробуйте его развернуть. Некоторые оптимизирующие компиляторы будут пытаться развернуть циклы, но гораздо практичнее это сделать вручную (мы подробно рассмотрим методику разворачивания циклов в восемнадцатой главе «Техника оптимизации»).

Наконец, для фрагментов, наиболее критичных по скорости выполнения, трудно найти замену ассемблеру. Однако характерно, что только две-три подпрограммы действительно нуждаются в применении этого языка.

Ваша стратегия должна выглядеть так:

- При разработке новых алгоритмов пишите функции на Си;
- Занимайтесь их отладкой до тех пор, пока не убедитесь, что они правильно работают и соответствуют тем задачам, для которых предназначены;
- Найдите наиболее критичные по быстродействию части программы и перепишите их на ассемблере.

Примечание

Вам понравилась графика, использованная в этой главе в демонстрационных целях? Я уверен, что понравилась. Не будет преувеличением сказать, что качественные рисунки облагораживают игру в целом. Вся графика, использованная в этой главе, была пожертвована автором шестнадцатой главы Дениз Тайлер. Спасибо Дениз!

ИТОГ

Существует множество различных способов моделирования параллакса. Методы, представленные в этой главе, не являются самыми быстрыми и наиболее элегантными. Однако они обладают тем достоинством, что они относительно просты в реализации и для понимания.

Уверен, что применение технических приемов из этой И других глав при разработке своих

собственных игр доставит вам немало удовольствия. Не бойтесь экспериментировать! Это истинное наслаждение.

ТЕХНИКА ОПТИМИЗАЦИИ

В прошлом году я просил у Санта Клауса подарить мне суперкомпьютер Cray XPM, но, как обычно, он не принес ничего. И если вы не будете использовать компьютер с фемисекундным циклом выполнения команд и террабайтами оперативной памяти, нам обоим придется примириться с ПК.

Следовательно, мы должны делать наши программы видеоигр настолько быстрыми, настолько возможно. Для этого мы должны постараться выжать из ПК каждую унцию его мощности. В данной главе мы охватим следующие темы, связанные с приемами оптимизации:

- Передача параметров;
- Глобальные переменные;
- Указатель и ценность псевдоимени;
- Использование регистров;
- Оптимизация компилятора;
- Развертка циклов;
- Бинарное умножение;
- Таблицы поиска;
- Математика с фиксированной точкой;
- Встроенный ассемблер;
- Предмет изучения;
- Оптимизация построения пикселя;
- Оптимизация изображения пикселя.

Введение

Прежде чем углубиться в изучение приемов оптимизации видеоигр для ПК, я хочу дать вам несколько советов. Когда вы оптимизируете свои программы, не пытайтесь разом провести полную оптимизацию. В первую очередь обратите внимание на те части, время выполнения которых критично и где встречается наибольшее количество циклов. Возьмите эти функции и работайте с ними до тех пор, пока их быстродействие не начнет вас удовлетворять.

Попытка оптимизации всей игры за раз, как правило, приводит к полному беспорядку, и вы никогда не сможете ни отладить программу, ни добавить что-либо в игру. Оптимизация должна быть взвешенной с точки зрения наличия других факторов. Если оптимизация функции принесет ей лишние три процента быстродействия, но при этом сделает ее вдвое запутаннее, то стоит попробовать найти другое место для оптимизации.

Не пытайтесь свалить все операторы, в кучу и свести десяток строк программы к одной. Ненавижу смотреть на выражения типа

```
* (x+(y++) >=*(& (x++)>>2)+(char far *)y;
```

Получили представление? Такая запись не только выглядит коряво с точки зрения Си, но и Codeview вряд ли поможет вам в отладке строк, подобных этой.

Поскольку мы разрабатываем видеоигры, я могу гарантировать, что 90% времени будет затрачено не на осуществление игровой логики, а на различные графические преобразования. Поэтому нужно сделать все функции рисования настолько быстрыми, насколько это возможно, а строки, управляющие игрой, имеет смысл оставить простыми и понятными. Функция, рисующая на экране точку, может представлять собой «черный ящик», а игровая программа в целом — нет. Оставляйте ее удобочитаемой. С помощью приемов оптимизации, которые мы изучим в этой главе, быстродействие практически любой функции может быть увеличено от 2 до 10 раз. Не говоря о том, что вы получите замечательные результаты, еще и произведете впечатление на друзей.

Передача параметров

Из второй главы «Основы языка ассемблерам (да и из собственного опыта) вы должны помнить, что передача параметров функциям не является свободной в полном смысле слова. Параметры должны быть помещены в стек, сделаны доступными через использование указателя базы и, наконец, взяты со стека. Если вам нужна функция, которая складывает 8 чисел, передаваемых как параметры, то потребовалось бы написать что-нибудь вроде этого:

```
int Add_Em_All(int nl,int n2,int n3,int n4,
            int n5,int n6,int n7,int n8)
{
return (n1+n2+n3+n4+n5+n6+n7+n8);
}
```

(Конечно, это не является реальной функцией. Я привел ее только в качестве наглядного примера.) После компиляции этой функции ее тело будет выглядеть примерно так:

```
clc
```

```
    mov ax,00h
    adc ax,n1
    adc ax,n2
    adc ax,n3
    adc ax,n4
    adc ax,n5
    adc ax,n6
    adc ax,n7
    adc ax,n8
```

Конечно, в самом начале приведенного фрагмента необходимо создать фрейм стека и уничтожить его в конце. Однако суть в том, что на восемь выталкиваний и извлечений параметров уходит немало времени. Если вы посмотрите на скорость выполнения команд PUSH и POP, то обнаружите, чти они отнимают в три раза больше тактов процессора, чем ADC. Как видите, в данном случае передача параметров отнимает больше времени, чем выполнение самой функции. Это показывает нам, что мы должны стараться передавать только те переменные, которые действительно необходимы.

Также никогда не передавайте структуры как значения. Если вы определите структуру, написав что-либо похожее на это:

```
typedef struct point_typ
{
int x[10],y[10],z[10];
}point,point_ptr;
```

сделаете вызов

```
Display(point point_1);
```

то в стек будет помещена целая структура! Это очень плохо. Чтобы избежать подобного, при передаче структур всегда применяйте ссылки на них, а «передачу значением» используйте только для целых и прочих стандартных типов данных Си.

Может быть, вы спросите: «Почему не использовать глобальные переменные вместо параметров?» Рассмотрим эту идею более внимательно.

Глобальные переменные

Мы пишем видеоигры, правила которых иногда должны изменяться. Всякий, кто где-нибудь учился, знает, что глобальных переменных стоит по возможности избегать. Конечно, всякое бывает, но лучше, когда их очень мало, и прекрасно, когда они вообще отсутствуют. Здесь приведена точка зрения одного крутого знатока в программировании игр; «Используйте глобальные переменные всегда, когда они помогают в увеличении быстродействия, но при этом сохраняйте чувство меры». К примеру, у нас есть набор функций, которые рисуют точки, изображают линии и окружности. Эти функции требуют передачи им различного количества параметров: для построения точки нужно знать две координаты, а для рисования линии — целых четыре. Что же касается цвета, то он, вероятно может быть одним и тем же как для линий, так и для окружностей. Почему же не сделать, чтобы каждая функция рисовала текущим глобальным цветом? С этой целью можно задать переменную и назвать ее, например, draw_color. Если вы измените текущий цвет и сделаете сотню вызовов функции, то при этом изменить цвет достаточно будет только один раз. В результате вы сумеете избежать порядка двухсот операций обмена со стеком.

Но учтите, что применение глобальных переменных может оказаться немного похожим на употребление наркотика: чем больше его принимаешь, тем больше он нужен. Хороший программист всегда может сбалансировать использование глобальных переменных и увеличить с их помощью быстродействие программы на 5, а то и 10 процентов.

Указатели и использование псевдоимен

Эту тему условно можно назвать тактическим приемом, которым пользуются некоторые программисты, в то время как многие о нем даже и не подозревают. Например, у вас есть фрагмент такой программы:

```
for(t == y->stars[index].left;
t < stars[index].left + 100; t++)
{
position = point->x + point->y -point->z;
pitch = point->x * point->y * point->z;
roll-= point->ang + sin(t) * point->ang;
}
```

Этот фрагмент хоть и выглядит достаточно компактным, но, тем не менее, и к нему может быть применен способ оптимизации, связанный с использованием псевдоимен. Вы видите, что в этом примере присутствует несколько команд, ссылающихся на указатель. Тактика, которой мы здесь воспользуемся, заключается в замене всех указателей, встречающихся более двух раз, простыми переменными. Вышеуказанная функция в результате может быть переписана так;

```
t1=y->stars[index].left;
x=point->x;
y=point->y;
z=point->z;
ang=point->ang;
for(t=t1;t<t1+100;t++)
{
position=x+y+z;
pitch=x*y*z;
roll=ang+sin(t)*ang;
}
```

Несмотря на то, что новая версия длиннее, она выполняется быстрее, так как содержит только пять ссылок вместо 800. Конечно, доступ к переменным x, y и z отнимает некоторое время, но порядок его величины меньше, чем ссылки на структуры и указатели. И этим нужно воспользоваться.

Использование регистров

В конечном счете, все программы, которые мы пишем на Си, будут переведены в последовательность машинных команд. После этого превращения некоторые из регистров общего назначения окажутся занятыми выполнением задачи, определяемой программой. Однако, мы не можем быть уверенными, что регистры будут использоваться вместо медленной стековой памяти. Чтобы быть уверенным, что функции используют регистры для переменной индекса (или какой-нибудь другой), давайте попробуем заставить компилятор по мере возможности делать это. Необходимо всегда помнить, что доступ к регистрам процессора во много раз быстрее, чем к оперативной памяти. Это происходит потому, что регистры находятся внутри ЦПУ, а память - нет. Мы можем использовать ключевое слово register, чтобы скомандовать компилятору использовать регистр. Как пример, напишем функцию, которая делает перестановку без регистров.

```
void Swap(int..num l,num 2)
{
int temp;
temp=num_1;
num_l=num_2; num_2=temp;
}
```

А теперь перепишем ее, используя регистр как временную переменную.

```
Swap(int num_1,num_2)
{
register int temp;
temp=num_1;
num_1=num_2;
num_2=temp;
}
```

Если компилятор может, то он будет использовать регистр в качестве регистровой переменной и программа увеличит скорость выполнения на 10-15 процентов. В связи с использованием ключевого слова register нужно учитывать два момента:

- Компилятор не создает регистры, он использует стандартные регистры ЦПУ;
- Иногда форсирование компилятора для использования регистров делает программу медленнее. Будьте осторожны. Обычно не стоит применять переменные типа register в маленьких функциях.

Теперь поговорим об основных приемах оптимизации компилятора.

Оптимизация компилятора

Фирма Microsoft уверяет, что ее компилятор является оптимизирующим. Это предельно правдивое высказывание. Несмотря на то, что оптимизатор иногда в состоянии сделать вашу программу медленнее или даже привнести ошибки, он проводит классическую оптимизацию (с точки зрения ученых-компьютерщиков). Однако мы, как создатели игр, не можем доверять автоматической оптимизации. Следовательно, мы никогда не будем использовать никакие опции оптимизатора (хорошо, может быть только некоторые из них). Стоит попробовать поиграть с ними

только когда ваша видеоигра уже полностью готова и свободна от ошибок. Но ни в коем случае не рассчитывайте на оптимизацию во время разработки программы. Это не панацея. Единственное, что может принести реальную пользу, так это опция отключения контроля переполнения стека.

Обычно компилятор в начале любой процедуры вставляет небольшие фрагменты кода, называемые прологом, которые служат для проверки достаточности стекового пространства для размещения локальных переменных. Но если установить размер стека в несколько килобайт, то у вас никогда не будет проблем. Поэтому вы можете выключить контроль переполнения стека, что немного сократит время обращения к функции (для этого можно использовать директиву компилятора -GS).

Я считаю, что оптимизатор на самом деле может доставить хлопот больше, нежели разрешить проблем. Вскоре я специально напишу эффективную программу, чтобы доверить ее оптимизатору и посмотреть, что из этого получится. Иногда оптимизатор может без зазрения совести внести в вашу программу пару-тройку ошибок, и это является еще одной причиной, почему им не стоит пользоваться слишком активно. (Я уже задокументировал дюжину ошибок в новом компиляторе Microsoft — Visual C/C++ 1.5, так что это высказывание имеет под собой серьезное обоснование. Обычно я до сих пор использую C/C++ 7.0, находя его более надежным.)

Развертка циклов

Это очень старо. В далеком прошлом, в конце 70-х годов, когда Apple безраздельно правил миром ПК, королем среди процессоров считался процессор 6502. Люди всегда находили интересные пути, чтобы заставить его работать быстрее. Один из трюков, который был найден, называется *разверткой циклов.* Это технический прием, где программист на самом деле отменяет структуру цикла вручную, разбивая саму задачу цикла. Структура цикла сама по себе имеет небольшой заголовок и мы можем использовать это в наших целях. К примеру, мы хотели бы инициализировать поле men в 10000 структур. Можно было бы поступить так:

```
for (index=0;index<10000;index++)
{
player[index].men=3;
}
```

и это будет прекрасно работать. Однако переменная index инкрементируется и сравнивается 10000 раз. Поскольку цикл повторяется 10000 раз, то это означает, что будет 10000 переходов типа NEAR. Мы можем развернуть цикл и получить небольшой выигрыш в скорости его выполнения. Вот что мы могли бы сделать:

```
for(index=0;index<1000;index+=10)
{
player[index].men=3;
player[index+1].men=3;
player[index+2].men=3;
player [index+3].men=3;
player[index+4].men=3;
player [index+5].men=3;
player [index+6].men=3;
player[index+7].men=3;
player [index+8].men=3;
player[index+9].men=3;
}
```

Теперь цикл выполняется только 1000 раз. Следовательно, index изменяется и сравнивается только 1000 раз, и при этом выполняется только 1000 переходов типа NEAR.

Этот пример написан мною еще и для того, чтобы показать вам, что развертка цикла может иметь и отрицательные стороны. Посмотрите внимательно на новую программу. Здесь к каждому индексу в каждой последующей операции присваивания добавляется смещение, и время, которое уйдет на это, может свести на нет выгоду от разворачивания циклов. Однако, чтобы исправить это, мы можем применить маленькую хитрость. Введем для индексирования структуры вторичную переменную new_index, которую будем увеличивать после каждого присваивания. Это приведет к увеличению скорости. Взгляните:

```
new_index=0;
for(index=0;index<1000;index+=10)
{
player[new_index++].men=3;
player[new_index++].men=3;
player[new_index++].men=3;
player[new_index++].men=3;
```

```
player[new_index++].men=3;
player[new_index++].men=3;
player[new_index++].men=3;
player[new_index++].men=3;
player[new_index++].men=3;
player[new_index++].men=3;
}
```

Новая программа работает быстрее, чем старая. Неплохо? Развертка циклов настолько эффективна, что у вас может возникнуть желание постоянно прибегать к этой уловке. Однако и здесь нужно знать меру. Дело в том, что машинные команды кэшируются внутри современных CPU, и слишком «массированное» разворачивание циклов может привести к проблемам переполнения кэша. Но если вы пользуетесь этим бережно (то есть ограничиваетесь тремя-восемью итерациями), то должны получить хорошие результаты.

Теперь поговорим о другом старом трюке - использовании операций сдвига для перемножения чисел.

Бинарное умножение

Впервые мы столкнулись с этим трюком в пятой главе, «Секреты VGA-карт». На ПК (да и вообще почти на любом компьютере на этой планете) система двоичных чисел используется для представления чисел в компьютере (хотя, я слышал и о «троичных» компьютерах). Поскольку разряды двоичных чисел являются степенью двух и каждое число помещается в слове как набор двоичных цифр, сдвиг слова влево или вправо смещает каждый его разряд на соседнее место. Эти операции автоматически удваивают число или делят его на два, соответственно. Взгляните на рисунок 18.1, чтобы увидеть это на примере.

Рис. 18.1. *Сдвиг битов для умножения и деления.*

Сдвигая число влево, вы умножаете его каждый раз на 2. Проделав это четыре раза, вы умножите его на 16, пять раз — на 32. Таким путем мы можем умножить число на любую степень двух, но как насчет других'чисел, таких, например, как 26? Для выполнения этого мы разобьем умножение на группы умножений по степеням двух. Число 26 может быть представлено как 16+8+2. Таким образом, если мы умножим произвольное значение X на 16, добавим к нему X, умноженное на 8 и, наконец, добавим X, умноженное на 2, то ответ должен быть таким же, как если бы мы умножили на 26. Взгляните:

```
Y=X*26=X*16+X*8+X*2=X<<4+X<<3+X<<1
```

Кстати, именно так поступают ученые-ракетчики,которые прекрасно знают, что сдвиг гораздо быстрее умножения, и мы также можем выиграть кучу времени, используя эту «реактивную» технику. Увеличение скорости вычислений достигается за счет того, что сдвиг — простая бинарная операция, в то время как умножение является действием гораздо более комплексным.

В связи со сказанным стоит упомянуть еще вот о чем;

- Во-первых, деление также может быть выполнено путем сдвига числа, но только вправо. Правда, деление сдвигом выполняется не так часто, как умножение, поскольку делитель разбить на составляющие обычно бывает значительно сложнее.
- Во-вторых, эта техника работает только с целыми числами без плавающей запятой. Значения типа FLOAT и DOUBLE хранятся в памяти в так называемом IEEE-формате, и их сдвиг приведет к переполнению разрядной .сетки.

Теперь мы можем перейти к следующей технике оптимизации — к таблицам поиска. Что ж, поищем их...

Таблицы поиска

Таблицы поиска, как следует из их названия, служат для поиска некоторых фактов. С их помощью можно искать что угодно. Суть применения таблиц поиска состоит в том, что вместо выполнения расчетов в процессе работы программы, мы предварительно выполняем все возможные вычисления, в которых может возникнуть необходимость, и сохраняем их в гигантской таблице. Далее, во время выполнения программы, мы смотрим в таблицу, используя параметр вычисления для поиска в ней конечного результата. Наиболее классическое использование справочных таблиц — предварительное вычисление трансцендентных функций, таких как синус, косинус, тангенс и т. д., поскольку они отнимают много времени для вычисления, даже с математическим сопроцессором.

Приведу пример использования справочных таблиц. Скажем, в программе нам необходимо вычислять синус и косинус угла, который может быть любым от 0° до 360° и является целым (то есть у нас не будет вычислений для углов с десятичной частью, таких как 3.3). Следующая программа создает справочную таблицу значений синусов и косинусов:

```
float sin_table[360],cos_table[360] ;
for(index=0;index<360;index++)
{
sin_table[index]=sin(index*3.14159/180);
cos_table[index]=cos (index*3.14159/180) ;
}
```

Этот программный фрагмент создает две таблицы, содержащих по 360 заранее вычисленных значений синусов и косинусов.

Теперь посмотрим, как использовать справочные таблицы, К примеру взглянем на это выражение:

```
x=cos(ang*3.14159/180)*radius;
y=sin(ang*3.14159/180)*radius;
```

Используя наши новые справочные таблицы, мы могли бы иметь:

```
x=cos_table[ang]*radius;
y=sin_table[ang]*radius;
```

Применение справочных таблиц может значительно увеличить быстродействие ваших игр. Единственная сложность заключается в том, что они отнимают много места. К примеру, игра Wing Commander использует справочные таблицы, которые содержат предварительно вычисленные виды кораблей для всех углов поворота. DOOM использует справочные таблицы, чтобы помочь в вычислении видимого расположения всех объектов игрового пространства.

При построении справочной таблицы учитывается все, что игрок может увидеть на экране. Необходимые для графических построений данные вычисляются перед началом игры или загружаются в готовом виде с диска. Затем, во время игры справочные таблицы получают привязку к окружающему игровому пространству и если табличные данные сообщают, что некоторую часть изображения невозможно увидеть, то графическая система устранения скрытых поверхностей удалит их.

Я сторонник идеи, что полноценная видеоигра может быть сделана только с огромным количеством справочных таблиц, и лишь небольшим программным кодом, отвечающим за осуществление игровой логики, а то и вовсе без такового. Как и в электротехнике, в видеоиграх используются конечные автоматы (с которыми мы столкнулись в тринадцатой главе, «Искусственный интеллект»). Однако вместо использования алгоритмов, моделирующих поведение объекта для следования из одного состояния в другое (как мы делали в тринадцатой главе), для управления существами можно привлечь справочные таблицы.

Этим техническим приемом в вычислительной технике пользуются для конструирования КА и мы также можем делать это в наших программах. Все что нам необходимо, это таблица, которая содержит текущее состояние и следующие состояния.

Итог: справочные таблицы великолепны. Применяйте их во всех случаях компьютерной жизни пока позволит объем памяти. Кроме того, когда вы составляете справочные таблицы, попробуйте, где это только возможно, использовать симметрию для уменьшения размера таблицы. Например, синус и косинус

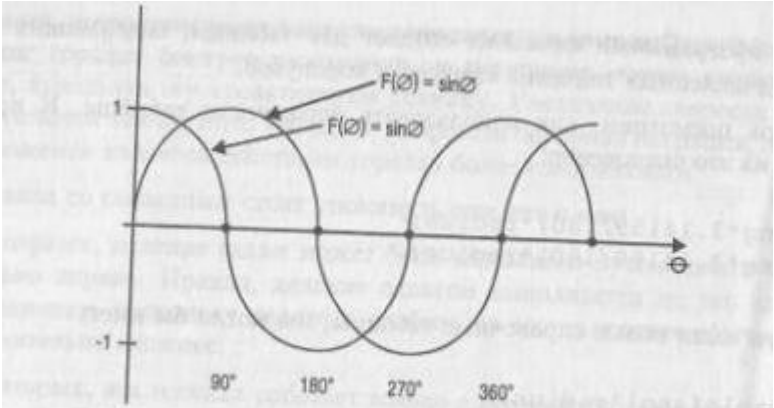


Рис. 18.2. Различие между функциями синуса и косинуса.

по сути являются одной и той же функцией с различием в 90о. Взгляните на рисунок 18.2. Мы видим, что синус и косинус выглядят почти одинаково. Фактически, они связаны следующими формулами:

`sin (angle)=cos(angle-90) cos (angle)=sin(angle+90)`

Поэтому мы могли бы создать единственную справочную таблицу для значений косинусов, а потом, когда нам понадобится вычислить синус угла, достаточно к исходной величине угла добавить 90е и использовать новый результат как индекс. Конечно, при этом нужно быть осторожным — новый угол может оказаться больше 360°. В этом случае вы должны выполнить его циклический возврат к нулю:

`if(angle>360) angle=angle-360;`

Для, доказательства того, что справочные таблицы могут увеличить скорость выполнения программ, я создал программку, которая использует как обычные функции sin и cos, так и справочную таблицу, хранящую заранее вычисленные значения синусов и косинусов, изображающую 100 окружностей. Программа, показанная в Листинге 18.1, начинается с заполнения таблиц. Затем она чертит 1000 окружностей, используя внутренние функции. Потом она ждет нажатия клавиши, после чего рисует 1000 окружностей, используя табличные данные. Запустив программу, вы обязательно почувствуете разницу. Также обратите внимание, что эта программа достаточно компактна и очень эффективна с точки зрения вывода графики.

Листинг 18.1. Сравнение выполнения программы с использованием справочных таблиц и встроенных функций sin и cos (LOOKNUP.C).

```
#include <math.h>
#include <stdio.h>
#include <graph.h>
float sin_table[360], cos_table[360];
main()
{
int index, x,y,xo,yo,radius,color,ang;
char far *screen = (char far *)0xA0000000;
// использовать библиотеку Microsoft для перехода
// в режим 320x200x256
_setvideomode(_MRES256COLOR);
// создать таблицы быстрого доступа
for (index=0; index<360; index++)
{
sin_table[index]= sin(index*3.14159/180} ;
cos_table[index]= cos(index*3.14159/180);
}
// нарисовать 1000 окружностей, используя встроенные
// функции sin и cos
for (index=0; index<1000; index++)
(
// получить случайные числа
radius = rand()%50;
xo = rand()%320;
yo = rand(}%200;
color = rand()%256;
for (ang=0; ang<3 60; ang++)
{
x = xo + cos(ang*3.14159/180) * radius;
У = yo + sin(ang*3.14159/180) * radius;
```

```
// нарисовать точку окружности
screen[(y<<6) + (y<<8) + x] = color;
}
}// все, ждать пока пользователь нажмет клавищу
printf("\nHit,a key to see circles drawn twith look up tables.");
getch();
_setvideomode(_MRES256COLOR);
// нарисовать 1000 окружностей, используя таблицы поиска
for (index=0; index<1000; index++)
{
// нарисовать случайную окружность
radius = rand()%50;
xo = randO %320;
yo = rand()%200;
color = rand()%256;
for (ang=0; ang<3 60; ang++)
{
x = xo + cos table[ang] * radius;
y = yo + sin_table[ang] * radius;
// нарисовать точку окружности
screen[(y<<6) + (y<<8) + x] = color;
} }
// подождать, пока пользователь нажмет любую клавишу
printf("\nHit any key to exit."};
getch();
_setvideomode(_DEFAULTMODE);
}
```

После запуска LOOKNUP.C вы должны согласиться, что справочные таблицы крайне удобны и могут здорово увеличить скорость выполнения программы.

Следующая тема будет касаться математики с фиксированной запятой.

Математика с фиксированной запятой

Математика с фиксированной запятой? Нет, это не новая точка зрения на дробные числа. Просто это немного другой путь рассмотрения компьютерной математики.

Существуют две формы математики, предназначенные для компьютера:

• Математика целых чисел;

• Математика с плавающей запятой.

Первая использует значения типов CHAR, INTEGER, LONG и т. д. Вторая оперирует числами FLOAT, DOUBLE и т. п. Разница между двумя этими видами математики заключается в том, как представлены числа в памяти и какие именно числа - целые или дробные - принимают участие в расчетах. Целые числа представлены в компьютере непосредственно в двоичной форме, без какого-либо кодирования. Как вы знаете, они могут быть как положительными, так и отрицательными, и у них отсутствует дробная часть. С другой стороны, числа с плавающей запятой должны иметь десятичные части.

Но к чему такая забота по поводу чисел? Объясняю: ПК может выполнять математические вычисления весьма быстро, но «весьма быстро» еще не значит «достаточно быстро для видеоигр». Даже с математическим сопроцессором ПК до сих пор оставляет желать лучшего в режиме реального времени при работе с трехмерной графикой. Известно, что вычисления с целыми выполняются гораздо быстрее, чем с дробными числами.

Вы можете спросить: «Почему мы обязательно должны использовать числа с плавающей запятой?» Ответ заключается в том, что по природе того типа программирования, которым мы занимаемся (трехмерная компьютерная графика), мы оказываемся перед неизбежной необходимостью достижения максимальной точности в наших вычислениях. Это заставляет нас использовать и дробные числа тоже.

Вычисления с плавающей запятой пожирают так много времени из-за способа представления чисел, с которыми они оперируют. Эти числа не являются в полном смысле двоичными, напротив, они хранятся в специальном формате IEEE, в котором характеристика (целая часть) и мантисса (экспонента) представлены в жутко свернутой форме, и прежде чем число использовать в вычислениях, его нужно еще расшифровать. Числа — это только инструмент для представления игровых объектов. Если бы вы захотели, то могли бы разработать свои собственные форматы хранения десятичных чисел. Это та область, где в бой вступает математика с фиксированной запятой. Например, можно представить число содержащим как целую, так и десятичную часть внутри отдельного целого. Как это сделать? Притворимся, что десятичные части существуют где-то внутри целого и что двоичные цифры слева являются целой частью числа, а все, что находится справа, это десятичная часть. Рисунок 18.3 должен помочь вам представить это.

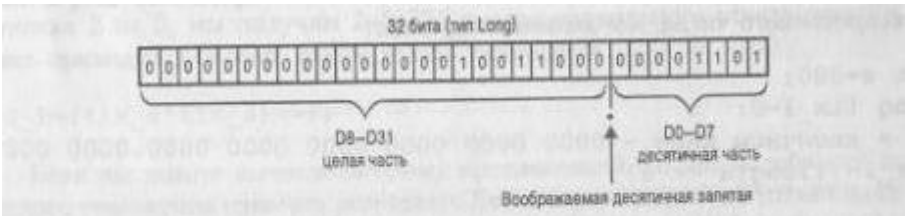


Рис. 18.3. *Представление числа 152.05 в виде 24 битов целой части и 8 битов десятичной.*

Где именно вы поместите десятичную часть, зависит от вас. Важно, чтобы позиция десятичной точки и выбор базового типа данных удовлетворяли нашим потребностям. Я предлагаю использовать тип данных LONG, который имеет 32 бита точности, а десятичную точку поместить где-нибудь посередине. Скажем, отведем для десятичной части восемь младших битов, тогда целая часть займет 24 старших разряда. Этого более чем достаточно для наших потребностей, поскольку в видеоиграх не нужна очень высокая точность. Пары знаков после запятой будет вполне достаточно. Чтобы использовать математику с фиксированной запятой, нам необходимо только разобраться, как выполнять с ней несколько операций:

- присваивание;
- сложение;
- вычитание;
- умножение;
- деление;
- сравнение.

Прежде чем разобраться в способах осуществления этих операций, мы все-таки должны понять как объявить число с фиксированной запятой. Взгляните, это действительно трудно:

```
long fix_1,fix_2,fix_3;
```

Что, обманул вас на секунду, да? Не правда ли, проще не придумаешь? Как я сказал ранее, мы используем тип данных LONG для чисел с фиксированной точкой и только предполагаем наличие десятичной части. И это все, что нам нужно для определения числа с фиксированной запятой.

Присваивание

Теперь поговорим о присваивании. Если мы хотим присвоить целую часть фиксированного числа, мы делаем .следующее:

```
int a=300;
long fix_1=0;
// в двоичном виде - 0000 0000 0000 0000 0000 0000 0000 0000
fix_1=((long)a << 8);
```

Сложнее обстоит дело с присваиванием дробных чисел. Для этого мы должны использовать умножение с плавающей запятой и записать число в LONG. Ниже показано, как выполнить такую операцию присваивания:

```
long fix_1 = (long) (23.4*256)
```

Мы умножаем на 256 потому, что это эквивалентно сдвигу на восемь разрядов влево (помните, я уже говорил, что нет смысла сдвигать числа с плавающей запятой).

Сложение и вычитание

Чтобы складывать или вычитать числа с фиксированной запятой, мы можем использовать обычные операторы Си. К примеру, чтобы сложить два числа и записать результат в третье, мы могли бы сделать так:

```
fix_3=fix_1+fix_2
```

Вычитание получается точно так же. Кроме того, вы можете использовать и отрицательные значения, поскольку внутреннее представление типа данных LONG учитывает знак, и это вполне применимо также и к числам с фиксированной запятой.

Умножение

Самая сложная из всех операций — это умножение. Здесь мы должны соблюдать осторожность: существует несколько нюансов, которые могут наплодить множество ошибок. Например, когда умножаются два числа с фиксированной запятой, для сохранения результата может потребоваться в два раза больше битов, нежели содержат сомножители. Другими словами, если оба сомножителя были 32-битными, не исключена возможность получить 64-битный результат. То есть мы должны

следить за величиной чисел с фиксированной запятой, которые перемножаются. Как и обычно, для этого мы будем использовать оператор умножения. Однако когда умножение выполнено, мы должны сдвинуть результат на 8 позиций назад вправо. Это нужно сделать потому, что когда мы задаем число с фиксированной запятой, то искусственно перемножаем его на 256 (помните, восемь младших разрядов заняты под десятичную часть). Поэтому мы должны сдвинуть окончательный результат назад восемь раз вправо, иначе говоря, результат должен быть разделен на 256. Если этого не сделать, то, умножая 2 на 5, мы получим 2x5x256 вместо правильного ответа равного 10. Здесь приводится способ, каким делать умножение.

```
fix_1=(fix_2*fix_3)>>8;
```

Если вы хотите вычислить сумму произведений, то нет надобности после каждого умножения сдвигать результат. Достаточно сделать это только один раз в самом конце расчетов. Рассмотрим пример:

```
fix_1=(fix_2*fix_3+fix_4*fix_5)>>8;
```

Это свойство чисел с, фиксированной запятой могло бы пригодиться, если вы захотите оптимизировать приведенный фрагмент и дальше с целью избавиться от всех сдвигов.

Деление

При выполнении деления я предлагаю вместо использования символа деления умножать на обратную величину. Как правило, это несложно сделать. Запомните, что деление — всегда более медленная операция, чем умножение, независимо от того, применяете вы числа с фиксированной или с плавающей запятой. Здесь приводится пример того, как могло бы быть выполнено деление:

```
fix_1=(long)(256*1/34);
fix_2=(fix_3*fix_1)>>8 ;
```

Прежде чем мы перейдем к следующей теме, мне бы хотелось затронуть некоторые детали, о которых прочие авторы обычно не любят говорить. Это точность и максимальное цифровое представление.

Точность

Поскольку мы договорились, что в нашем формате чисел с фиксированной запятой восемь младших разрядов будут содержать десятичную часть, то самым маленьким числом, которое можно представить, окажется значение 1/256 или примерно 0.004. Следовательно, максимальная ошибка будет получаться при умножении двух чисел. Наибольшее число, которое мы можем получить в произведении, равно 32761. Следовательно, наибольшая ошибка, которая может закрасться в расчеты, это 0.004x32761 или 131.044. Ого! Это слишком много. Однако в действительности у нас никогда не будет ошибок такой величины. Только вы не должны с одним и тем же числом выполнять больше 2-5 умножений, и сомножители не должны превышать 32761. Как правило, в большинстве случаев ошибки не будут превышать 0.01-0.5, что вполне допустимо, поскольку 90 процентов всех расчетов направлены на определение местоположения пикселей на экране и результаты все равно округляются.

Хватит насчет точности. Перейдем к определению максимального числа, которое может быть представлено в нашей системе с фиксированной запятой.

Максимальное цифровое представшие

Поскольку у нас есть 24 бита для целой и 8 бит для десятичной части числа, вы можете подумать, что таким образом можно представить значения вплоть до 2^{24} или 16777216. Почти, но не совсем. Так как число с фиксированной запятой может быть и положительным и отрицательным, мы располагаем числами в диапазоне от -8388608 до +8388608. Мы можем без всяких проблем складывать и вычитать числа из этого диапазона, но при умножении, должны быть исключительно осторожны, чтобы не переполнить тип LONG.

Когда я изучал математику с фиксированной запятой в первый раз и пытался алгоритмизовать ее, то допустил ошибку. Я использовал схему, похожую на нашу (то есть 24 бита для целой части и 8 бит для десятичной) и думал, что наибольшие числа, пригодные для умножения, могут быть любыми, лишь бы результат укладывался в 24 бита. Это означало бы, что можно перемножить 4096 на 4096 и получить правильный ответ. Ошибочка! Я забыл об остальных 8 битах десятичной части- Следовательно, в действительности я умножил 4096x256x4096x256, что составляет примерно 1.09x1012. Поскольку тип LONG состоит из 32-х битов, то он может представлять числа от -2147483648 до +2147483648 (которые в 1000 раз меньше полученного результата). Мораль сей басни такова, что числа с фиксированной запятой остаются принадлежащими типу LONG, и если в них записать числа, интерпретируемые как LONG, то при умножении возникнет переполнение.

Наибольший результат умножения, который может быть получен в нашей системе с

фиксированной запятой равен 32761 или 181 в степени 2. Число 181 было получено исходя из следующих соображений: это число, которое, будучи умноженным на 256 и возведенным в квадрат, не должно превышать размерности типа LONG (+2147483648). Мы используем 256, так как у нас есть восемь двоичных цифр, а 2^8 равно 256.

Как, ни странно, но наибольшее число, которое может быть получено в результате умножения — это 32761, а величина чисел для сложения может достигать примерно 8000000? Ну что ж... где-то мы нашли, а где-то потеряли.

Чтобы помочь вам в экспериментах с числами с фиксированной запятой и увидеть некоторые их интересные свойства, я создал небольшую библиотеку и программу main() для демонстрации их использования (Листинг 18.2). Я предлагаю вам потратить немного времени, чтобы получить действительно целостное понимание сути чисел с фиксированной запятой, поскольку это очень важно и про это мало кто знает. Никогда не надо забывать, что десятичная запятая является воображаемой!

Листинг 18.2. Функции библиотеки системы с фиксированной запятой (FIX.C).

```
// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ////////////////////////////////
#include <math.h>
#include <stdio.h>
// определим наш новый тип чисел с фиксированной запятой
typedef long fixed;
//ФУНКЦИИ //////////////////////////////////////////
fixed Assign_Integer(long integer)
{
return((fixed)integer << 8);
} // конец функции присваивания целочисленного
значения ////////////////////////////////////////////////////////
fixed Assign_Float(float number)
{
return((fixed)(number * 256)};
} // конец функции присваивания значения с плавающей
запятой /////////////////////////////////////////////////////////
fixed Mul_Fixed(fixed fl,fixed f2)
{
return ((fl*f2) >> 8);
} //конец функции умножения ///////////////////////////////////////////////////////////////
fixed Add_Fixed(fixed fl,fixed f2)
{
return(f1+f2);
} // конец функции сложения ////////////////////////////////////////////////////////////////
Print_Fixed(fixed fl)
{
printf("%ld.%ld", f1>>8, 100*(unsigned long) (f1 & 0x00ff)/256);
} // конец функции вывода числа с фиксированной запятой
// ОСНОВНАЯ ПРОГРАММА ////////////////////////////////////
main(}
{
fixed f1,f2,f3;
fl = Assign_Float(15);
f2 = Assign_Float(233.45);
f3 = Mul_Fixed(f1,f2);
printf("\nf1:");
Print_Fixed(f);
printf("\nf2:");
Print_Fixed(f2) ;
printf("\nf3:");
Print_Fixed(f3) ;
} // конец функции main
```

Конечно, в собственных программах вы фактически не будете использовать функции сложения и умножения. Я поместил их здесь только для того, чтобы вы могли поэкспериментировать со свойствами чисел с фиксированной запятой.

Теперь перейдем к встроенному ассемблеру.

Встроенный ассемблер

Ассемблировать или не ассемблировать: вот в чем вопрос. Почти шекспировская задача и нам ее сейчас предстоит разрешить.

Как я говорил ранее, вы должны использовать MASM и встроенный ассемблер, только если в этом действительно возникает необходимость. Ведь все равно операции, которые вы выполняете, являются машинно-зависимыми по своей природе. Вам не нужен ассемблер ни для чего, кроме создания графики и звука. Ни в коем случае не стоит использовать ассемблер для реализации игровой логики и алгоритмов. Кроме того, если ассемблер вам необходим для увеличения быстродействия, применяйте вместо MASM встроенный ассемблер. Он проще в использовании и его применение отнимает меньше времени при разработке программ. Мы обсудим несколько методов, которые увеличат скорость выполнения ваших программ во много раз.

Теперь рассмотрим некоторые из функций, которые мы написали в предыдущих главах и оптимизируем их с помощью новых технических приемов.

Оптимизация рисования пикселей

Первое, что мы должны сделать, это насколько возможно оптимизировать по быстродействию функцию рисования пикселей. Ведь на ней базируются, все остальные графические построения! Давайте возьмем функцию Plot_Pixel() из Листинга 5.4 (вы найдете ее в пятой главе «Секреты VGA-карт») и поглядим можем ли мы оптимизировать ее дальше. Листинг 18.3 содержит новую функцию.

Листинг 18.3. Функция для быстрого рисования пикселей.

```
void Plot_Pixel_Fast(int x,int y,unsigned char color)
(
// Функция рисует на экране точку несколько быстрее
// за счет использования сдвига вместо умножения:
// 320 * y = 256 * y + 64 * y = y << 8 + y << 6
video_buffer[((y<<8}+(y<<6) )+x]=color;
}
```

Все. С точки зрения Си эта функция уже оптимизирована настолько, насколько возможно. Я могу вам предложить только следующее:

- Перепишите программу на встроенном ассемблере;
- Не передавайте параметры;
- Возможно, создайте справочную таблицу из 64000 элементов, каждый из которых содержит адрес видеобуфера, соответствующий координатам X и Y.

Правда, я думаю, что использование справочной таблицы на самом деле только замедлит выполнение программы. Ведь операция индексирования таблицы может отнять больше времени, чем два сдвига и сложение, необходимые для вычисления адреса. Но вот первые два пункта заслуживают внимания. Начнем с использования глобальных переменных, заменяющих передачу параметров. Например, определим такие глобальные переменные:

```
int plot_x, plot_y, plot_color;
```

Тогда нашу функцию можно переписать, как показано в Листинге 18.4

Листинг 18.4. Другая версия функции построения пикселя.

```
void Plot_Pixel_Global(void)
{
video_buffer[((plot_y<<8) + (plot_y<<6))+plot_x]=plot_color;
}
```

Это уже будет работать значительно быстрее, но нам все равно перед вызовом нужно выполнить операцию присваивания для переменных plot_x, plot_y и plot_color. Следовательно, вопрос в следующем - отнимет ли операция присваивания глобальных переменных меньше времени, чем передача параметров, создание и удаление фрейма стека вместе со ссылками на параметры? Может да, а может и нет. Все зависит от ситуации. Но цель заслуживает того, чтобы попробовать испытать и этот метод.

Теперь перепишем процедуру на ассемблере. Результат показан в Листинге 18.5

Листинг 18.5. Версия функции рисования пикселя на ассемблере.

```
Plot_Pixel_Asm(int x,int y,int color)
{
_asm{
```

```
les di,video_buffer   // загрузить регистр es значением
// сегмента видеобуфер
 mov di,y              // поместить в di y-координату пикселя
shl di,6              // умножить на 64
mov bx,di             // сохранить результат
shl di,2              // умножить еще на 8 (итого, на 256)
add di,bx             // сложить результаты
add di,x              // прибавить x-компонент
mov al,BYTE PTR color // записать цвет в регистр al
mov es:[di],al        // нарисовать пиксель
}
}
```

Ладно, покончим с этим. Ассемблерный вариант работает всего лишь на 2 процента быстрее, чем версия на Си. На то есть две причины:

- Во-первых, компилятор Си проделывает неплохую работу, транслируя программу в машинные коды. Хотя, как вы знаете, мы можем сделать это вручную, используя ассемблер;
- Во-вторых, когда вы используете встроенный ассемблер, он сохраняет все регистры и позже восстанавливает их. Единственный способ избавления от этого - написать внешние ассемблерные функции с использованием MASM. В данном случае это вполне допустимо, поскольку мы оптимизируем такую важную операцию как построение пикселя.

Наконец, я хочу показать вам последний пример оптимизации, которая позволяет ускорить вывод на экран в два раза.

Оптимизация изображаемой линии

Игры типа DOOM и Wolfenstein 3-D не используют все известные техники трехмерных графических преобразований, как это делают, например, имитаторы полетов. В них применяются совершенно гениальные методы для создания трехмерных образов. Эти методы базируются на изображении большого количества линий, проходящих в одном направлении. Обычно рисуются обычно вертикальные или горизонтальные линии и только в удаленных предметах присутствуют диагональные прямые. Следовательно, мы должны научиться максимально быстро проводить горизонтальные и вертикальные линии. Сейчас мы поговорим о горизонтальных прямых.

Как мы проходили в пятой главе «Секреты VGA-карт», для изображения горизонтальной линии лучше всего применить функцию memcpy () (см. Листинг 5.6). Начальный и конечный адреса вычисляются из X-координат крайних точек линии. Вернувшись назад к этому методу, приходится признать, что он не совсем совершенен, поскольку memcpyO для перемещения данных использует тип BYTE. Но профессионалы в программировании игр знают, что VGA-карта работает быстрее с данными типа WORD. Следовательно, нужно попытаться исправить этот недостаток и записывать в видеобуфер WORD вместо BYTE. Сейчас я собираюсь переписать функцию изображения горизонтальных линий, учитывая это пожелание.

Это не так легко, как кажется, потому что конечной целью должна быть забота о работоспособности. Вы видите, что когда вы пишите WORD в видеобуфер, вы в сущности строите два пикселя. Вы должны быть осторожны, принимая это во внимание во время рисования линий. Скажем, мы хотели изобразить линию, тянущуюся от точки (51,100) до точки (100,100). У нас получится линия, которая выглядит чем-то похожей на рисунок 18.4.

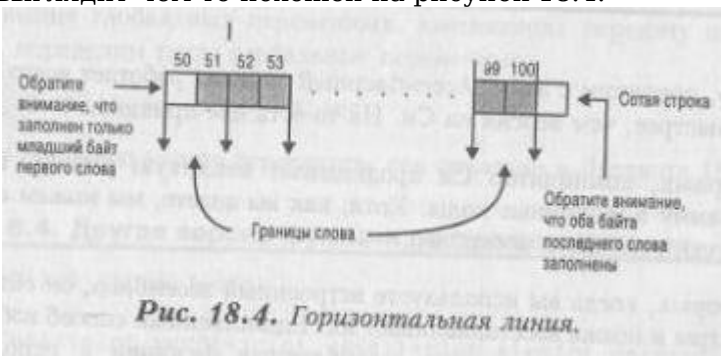


Рис. 18.4. Горизонтальная линия.

Завершим анализ изображения этой линии тем, что каждая конечная точка в действительности занимает один байт, а не два. Следовательно, мы должны написать программу, которая может управлять состоянием, когда конечные точки BYTE или WORD граничат. Программа из Листинга 18.6 делает это.

Листинг 18.6. Программа, изображающая горизонтальную линию и использующая для этого тип WORD (HLINEF.C).

```
H_Line_Fast(int xl,int x2,int y,unsigned int color)
{
unsigned int first_word, middle_word, last_word,line_offset,
index ;
// тестируем 1 бит начальной x-координаты
if ( (x1 & 0x0001))
{
first_word = (color << 8);
}
else
{
// заменить цвет в обоих байтах
first_word = ( (color<<8) | color) ;
}
// тестируем первый бит в x2
if( (x2 & 0x0001) )
{
last_word = ((color<<8) | color);
)
else
{
// поместить цвет только в старший байт
last_word = color;
}
// теперь мы можем рисовать горизонтальную линию,
```

```
// выводя сразу по два пикселя
line_offset = ( (y<<7) + (y<<5) ); // y*160, поскольку в линии 160 слов
// вычислить цвет в середине
middle_word = ((color<<8) | color);
// левая граница
video_buffer_w[line_offset + (x1>>1)]= first_word;
// середина линии
for (index=(x1>>1)+l, index<(x2>>l); index++)
video_buffer_w[line_offset+index] = middle_word;
// правая граница video_buffer_w[line_offset+(x2>>1)] = last_word;
}
```

В начале работы функция проверяет, находятся ли конечные точки на границе байта (BYTE)? Основываясь на результатах проверки, функция создает два слова (WORD): одно будет началом линии, а другое ~ концом. В зависимости от результатов начального теста эти два слова содержат либо один, либо два байта, представляющих цвет. Затем функция изображает линию. Это выполняется следующим образом:

- Пишется первый WORD, соответствующий левой границе;
- Создается цикл FOR, который предназначен для записи в WORD середины линии;
- Пишется WORD, соответствующий правой границе линии.

Хотя эта функция почти в два раза длиннее, чем первоначальная H_Line, она почти в два раза быстрее.

(Существуют, правда, небольшие накладные расходы при вычислении границы). Чтобы сделать ее еще быстрее, я мог бы переписать часть, которая изображает середину на встроенном ассемблере, но я думаю, что вы и сами сделаете это в качестве легкого упражнения.

ИТОГ

Мы узнали множество полезных вещей, и даже если вы никогда не начнете писать игры, определенно станете более умелым программистом, поскольку приобрели новые навыки. У вас появилось несколько мощных методов оптимизации. Только не думайте, что лучше этих методов не бывает или что предложенные способы оптимизации годятся на все случаи жизни.

Наше длинное путешествие в мир оптимизации подошло к концу, и я хочу закончить его приглашением в следующую, девятнадцатую главу, где мы, наконец, на примере игры Warlock применим все, чему уже научились. Итак, до следующей главы.

ИГРА WARLOCK (КОЛДУН)

Если вы добрались до этого места, у вас определенно неслабая воля. Вы преодолели множество трудностей, открыли кучу секретов и на страницах предыдущих глав узнали столь много разных вещей, что трудно даже представить, как все это вообще можно запомнить! Вы, должно быть, относитесь к тем, кому это удалось.

Я обещал дать готовую трехмерную игру. Но я вас обманул. Чуть-чуть. Если б я написал полную игру, вам ничего не осталось бы делать и вы не смогли бы ничему научиться. Поэтому я составил только скелет, ядро трехмерной игры, выполняющейся в реальном времени. Позже, с помощью этой книги вы и сами сможете написать все недостающие функции, чтобы закончить ее разработку.

Взгляните на рисунок 19.1. На нем показаны несколько кадров игры Warlock.

Постарайтесь превратить эту заготовку в настоящую игру. Может быть, у вас получится какая-нибудь простая игрушка типа «Захвата флага», а может быть, вы создадите нечто подобное DOOM'у. Что бы вы ни решили, мне будет приятно увидеть результат, поэтому в конце главы я сообщу адрес, по которому вы можете прислать любые пожелания или вопросы (я даже мог бы опубликовать вашу игру в какой-нибудь подходящей книге или журнале).

В этой заключительной главе мы рассмотрим составляющие части того игрового ядра, которое я создал для вас, чтобы вам легче было использовать и улучшать его. Мы коротко охватим следующие темы:

- Сюжет игры Warlock;
- Компоненты игры;
- Новый отсекатель лучей;
- Изображение текстуры;
- Оттенение;
- Использование ассемблера;
- Цикл игры;
- Игровое поле;
- Режим демонстрации;
- Размещение объектов в пространстве;
- Достижение некоторой скорости;
- Несколько слов напоследок.

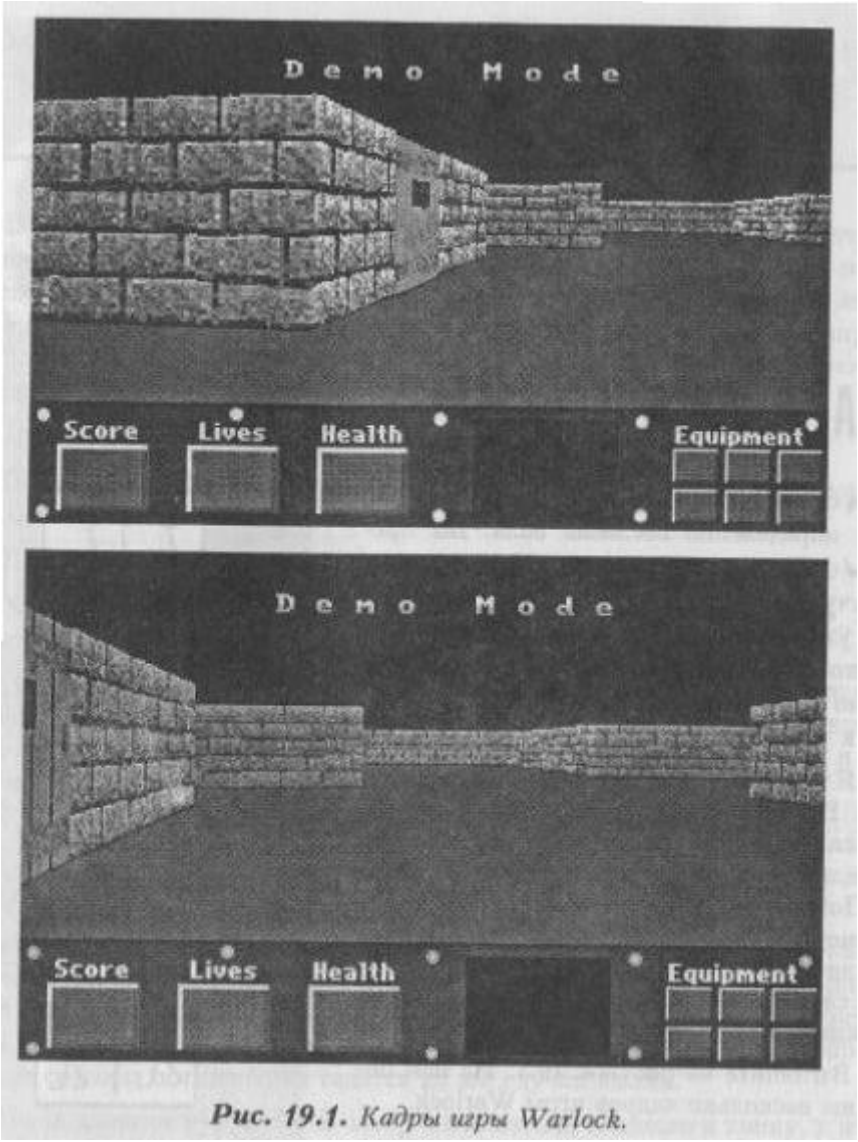


Рис. 19.1. *Кадры игры Warlock.*

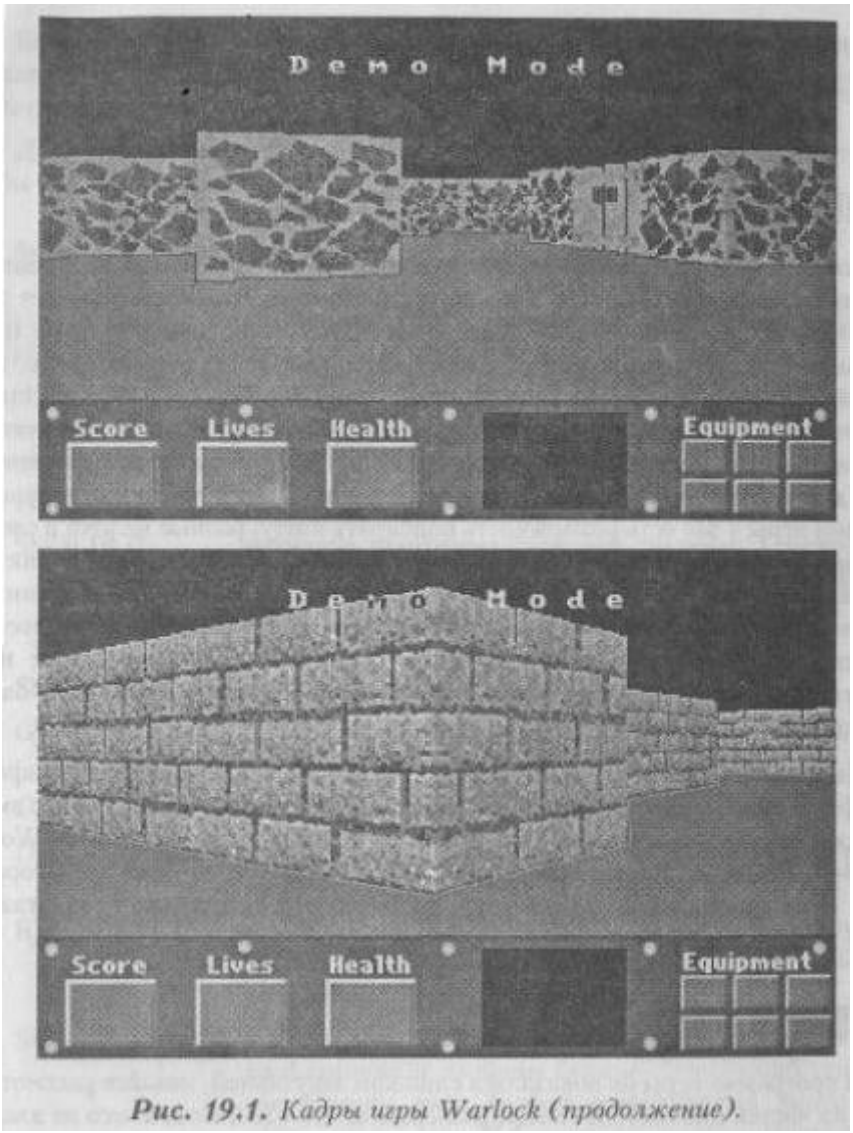


Рис. 19.1. Кадры игры Warlock (продолжение).

Сюжет игры

Первым делом при разработке игры нужно позаботиться о том, чтобы она не повторяла уже существующие. Можно предположить, что Warlock будет трехмерной игрой с поединками на мечах и волшебством. Основная идея будет такой:

главный герой прокладывает путь к своей цели с помощью магического оружия и устных заклинаний. Начав путешествие с голыми руками, он должен пройти три уровня, на каждом из которых спрятаны части сверхмощного магического оружия. Из этих частей он собирает волшебный меч, обладающий достаточной силой для уничтожения ужасного мертвеца, находящегося на последнем уровне. Во время игры у вас есть возможность подбирать пищу, разные мелочи и свитки. Мелочи могли бы восстанавливать здоровье персонажа, давать ему дополнительную силу, а в свитках он мог бы находить заклинания (произносимые синтезированным голосом). Эти заклинания служат для преодоления особо серьезных ситуаций, где бессильно другое оружие. Кроме того, в игре также могут присутствовать два-три типа летающих монстров, которые выглядели бы несколько лучше бегающих.

Игра должна неплохо выглядеть: несколько уровней с красивым графическим оформлением, пара монстров и изящные синтетические звуки. Все это вместе с трехмерным изображением должно поднять нашу игру до уровня Wolfenstein 3-D. Конечно, вы можете делать с ядром игры все, что вам заблагорассудится. Я не навязываю вам сюжетную линию Warlock, а только предлагаю.

Компоненты игры

Чтобы программа игры не показалась слишком запутанной, давайте рассмотрим, из каких частей она состоит. Затем разберемся, что в ней есть, а чего не хватает.

Примечание по звуку

Для получения синтетических звуковых эффектов в игре используется Sound Blaster, и чтобы услышать что-нибудь, вы должны настроить его соответствующим образом. При компиляции я

сделал такие установки:

- DMA#1
- IRQ #5
- I/O port-220h

Если ваши установки лучше, вы всегда можете заново оттранслировать исполняемый модуль и сделать соответствующие изменения в функции Voc в начале модуля main.

Даже если вы и не имеете звуковой карты, все равно запустите программу, чтобы увидеть ее в действии.

Загрузка и компиляция Warlock

Исполняемый модуль называется WARLOCK.EXE. Для запуска программы достаточно ввести warlock в командной строке DOS.

В программе игры используются три основных модуля плюс несколько процедур на ассемблере, осуществляющих рендеринг. Имена и функции модулей перечислены ниже;

WARLOCK.C исходная программа игрового ядра.

SNDLIB.C исходная программа звуковой системы. Соответствующие функции вы видели раньше, но для Warlock я создал более совершенный интерфейс и разместил его в этом файле.

SNDLIB.H файл заголовков для SNDLIB.C.

GRAPHICS.C окончательная графическая библиотека (почти идентичная GRAPHO.C).

GRAPHICS.H файл заголовков для GRAPHICS.C.

DRAWG.ASM эта функция изображает черное небо и серую поверхность земли, используя 32-битовые команды.

RENDER32.ASM эта функция копирует изображение из дублирующего буфера в видеопамять. Она также использует 32-битовые команды.

SLIVER.ASM эта функция изображает отдельный фрагмент текстуры изображения в двойном буфере. Это способ генерации трехмерного вида (набор вертикальных полосок). Я снова использовал 32-битовые команды, и таким образом вы получаете доступ к регистрам дополнительного сегмента.

После трансляции всех файлов они связываются вместе, используя warlock.c как главный модуль. Я делал вот как: сначала создал общую библиотеку, в которой объединил звуковую и графическую библиотеки, а также дополнительные функции на ассемблере. Затем я откомпилировал ее и скомпоновал с главной программой WARLOCK.C.

Учтите, что компиляция производится с использованием модели памяти MEDIUM. Также не забудьте использовать опцию /G2, позволяющую учитывать специфические особенности команд 286-го процессора. Все необходимое записано в стандартном BAT-файле, который я использовал для компиляции программы с помощью Microsoft C/C++ 7.00:

```
cl -AM -Zi -c -Fc -Gs -G2 %l.c
```

Эта строка переводится так: «компилировать для модели памяти MEDIUM, с использованием команд 286-го процессора, с отключенным контролем переполнения стека и генерировать листинг программы». Чтобы получить представление, как компонуется программа, взгляните на рисунок 19.2. На нем показана схема, по которой отдельные кусочки объединяются в единое целое.

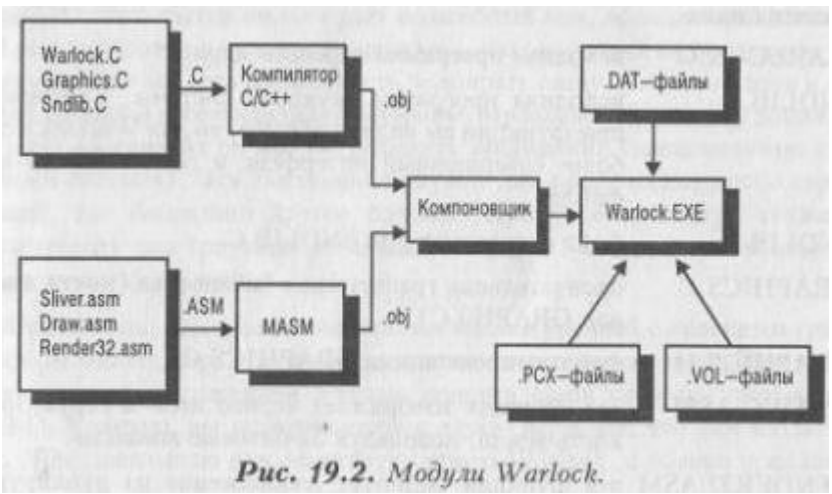


Рис. 19.2. *Модули Warlock.*

Режим демонстрации

Игра начинается в демонстрационном режиме, на это указывает надпись DEMO в верхней части экрана. Вы увидите, как демонстрационная программа «прокручивает» примерно минутную пробежку в пространстве. Когда надоест, нажмите клавишу Esc. Это позволит вам самим побегать по лабиринту.

Движение по игровому пространству

В этом режиме используйте клавиши управления курсором для перемещения в пространстве. Чтобы открыть дверь, нажмите клавишу Пробел.

Как видите, окружающая среда Warlock напоминает подземную тюрьму. Вы можете заметить, что в этом пространстве не существует никаких объектов. Также отсутствуют полы, потолки или тени — только стены, да кое-где двери. Мы обсудим все эти темы чуть позже в этой главе.

Когда вдоволь нагуляетесь, вновь нажмите Esc для выхода из игры. Вы пока побродите по закоулкам, а я подожду здесь.

Вернулись? Теперь поговорим немного по поводу каждой части игрового ядра, представляющей для нас хоть какой-нибудь интерес.

Новый отсекатель лучей

Отсекатель; лучей, который вы видели в шестой главе, «Третье измерение», по быстродействию был подобен матричному принтеру, пытающемуся напечатать полутоновую картинку с высоким разрешением! Словом, он был чрезвычайно медлительным. Так получилось потому, что он был написан совершенно прямолинейно и целиком на Си, в нем использовалась математика с плавающей запятой и не применялись справочные таблицы. Отсекатель лучей, который я создал для Warlock, немного другой. Новая программа оптимизирована во многих частях, хотя есть возможность ускорить ее работу еще примерно в два раза. Однако после этого отсекатель лучей мог бы стать таким жутким, что вы не захотели бы иметь с ним дело. (Я сам едва могу вообразить, как он выглядел бы в этом случае, а ведь я его написал!)

В конечном счете, быстродействие отсекателя лучей — не самое узкое место. Таким узким местом в гораздо большей мере является перемещение содержимого двойного буфера в видеопамять. Поэтому будет значительно лучше чередовать отсечение лучей с визуализацией изображения.

Мы поговорим немного больше по этому поводу и о некоторых других вещах немного позже, а сейчас перейдем непосредственно к программе нового отсекателя лучей (Листинг 19-1).

Листинг 19.1. Новая программа отсечения лучей.

```
void Ray Caster(long x,long y,long view_angle)
{
// Эта функция является сердцем системы. Она основана на программе
// RAY.C. Обратите особое внимание на оптимизацию по скорости
// работы. Это достигается, в основном, за счет использования таблиц
// поиска и применения математики с фиксированной запятой.
int   cell_x,      // Текущие координаты луча
cell_y,
ray,         // Обрабатываемый луч
casting=2,   // Указывает, обработаны ли обе координаты
x_hit_type, // Указывают на тип объекта, пересеченного
y_hit_type, // лучом. Используется при выборе текстуры
x_bound,     // Следующая точка пересечения луча
y_bound,
next_y_cell, // Используется при отрисовке ячейки, в
next_x_cell, // которой находится луч
xray=0,      // Счетчик проверенных точек пересечения
// с вертикалями плана
yray=0,  // Счетчик проверенных точек пересечения
// с горизонталями плана
x_delta,     // Смещение, которое необходимо добавить
y_delta,     // для перехода к следующей ячейке
xb_save,
yb_save,
xi_save,     // Используется для сохранения координат
yi_save,     // точек пересечения
scale;
long  cast=0,
dist_x,      // Расстояние ;от точки пересечения
dist_y;      // до позиции игрока
float xi,          // Используются при расчетах
yi;          // пересечений
// СЕКЦИЯ 1 ////////////////////////////////////////
// ИНИЦИАЛИЗАЦИЯ
// Рассчитываем начальный угол по отношению к направлению
```

```
// взгляда игрока. Угол зрения - 60 градусов. Обрабатываем
//поочередно обе половины угла
if ( (view_angle-=ANGLE_30) < 0 }
{
view_angle=ANGLE_360 + view angle;
}
// цикл по всем 320 лучам
for(ray=319;ray>=0; ray--)
{
// СЕКЦИЯ 2 ////////////////////////////
// Вычислить первое пересечение по X
// Нам необходимо узнать, какая именно половина плана,
// по отношению к оси Y, обрабатывается
if (view_angle >= ANGLE_0 && view_angle < ANGLE_180)
{
// Определяем первую горизонталь, которую может пересечь луч.
// На плане она должна быть выше игрока.
y_bound = (CELL_Y_SIZE + (y & 0xFFC0));
// Рассчитываем, смещение для перехода к следующей горизонтали
y_delta =- CELL_Y_SIZE;
// На основании первой возможной линии горизонтального
// пересечения рассчитываем отсечение по горизонтали
xi -= inv_tan_table[view_angle] * (y_bound - y) + x;
// Устанавливаем смещение ячейки
next_y_cell = 0;
} // Конец обработки else
{
// Рассчитываем первую горизонталь, которую может пересечь луч.
// На плане она должна быть ниже игрока
y_bound = (int)(y & 0xFFC0);
// Рассчитываем смещение для перехода к следующей горизонтали
y_delta = -CELL_Y_SIZE;
// На основании первой возможной линии горизонтального
// пересечения рассчитываем отсечение по горизонтали
xi = inv_tan_table[view_angle] * (y_bound - y) + x;
// Устанавливаем смещение ячейки
next_y_cell = -1;
} // Конец обработки // СЕКЦИЯ 3 ////////////////////////////
// вычислить первое пересечение по Y
// Нам надо знать, которая именно половина плана,
// по отношению к оси X, обрабатывается
if (view_angle < ANGLE_90 || view_angle >= ANGLE_270)
{
// Рассчитываем первое отсечение по оси Y
// Определяем первую вертикаль, которую может пересечь луч.
//На плане она должна быть справа от игрока
x_bound = (int)(CELL_X_SIZE + (x & 0xFFC0));
// Определяем смещение для перехода к следующей ячейке
x_delta = CELL_X_SIZE;
// На основании первой возможной линии вертикального
// пересечения вычисляем первое отсечение по оси Y
yi = tan_table[view_angle] * (x_bound - x) + y;
// Устанавливаем смещение ячейки
next_x_cell = 0;
}
else
{
// Определяем первую вертикаль, которую может пересечь луч.
// На плане она должна быть левее игрока
x_bound = (int)(x & 0xFFC0);
// Определяем смещение для перехода к следующей ячейке
x_delta = -CELL_X_SIZE;
// На основании первой линии вертикального пересечения
// рассчитываем первое отсечение по оси Y
yi = tan_table[view_angle] * (x_bound - x) + y;
// Устанавливаем смещение ячейки
next_x_cell = -1;
```

```
} // Конец обработки
//начать отсечение
casting       = 2; // Одновременно обрабатываем 2 луча
xray=yray=0; // Сбрасываем флаги пересечения
// СЕКЦИЯ 4 ////////////////////////////
// Продолжаем расчет для обоих лучей
while(casting)
{
if (xray!=INTERSECTION_FOUND)
{ // Рассчитываем текущую позицию для проверки
cell_x = ( (x_bound+next_x_cell) >> CELL_X_SIZE_FP);
cell_y = (int)yi;
cell_y>>=CELL_Y_SIZE__FP;
// Проверяем, не находится ли в проверяемой области блок
if ((x_hit_type = world[cell_y](cell_x])!=0)
{ // Рассчитываем расстояние
dist_x  = (long)((yi - y) * inv_sin_table[view_angle]) ;
yi_save = (int)yi;
xb_save = x_bound;
// Конец расчета по оси X
xray = INTERSECTION_FOUND;
casting--;
} // Конец обработки наличия блока
else
// Рассчитываем следующее пересечение по Y
{
yi += y_step[view_angle];
// Ищем следующую возможную X-координату пересечения
x_bound += x_delta;
} // Конец else
}
// СЕКЦИЯ 5 ////////////////////////
if (yray !=INTERSECTION_FOUND)
{ // Рассчитываем текущую позицию для проверки
cell x = xi;
cell_x>>=CELL_X_SIZE_FP;
cell_y = ( (y_bound + next_y_cell) >> CELL_Y_SIZE_FP) ;
// Проверяем, не находится ли в текущей позиции блок
if ((y_hit_type = world[cell_y][cell_x]) !=0)
// Вычисляем расстояние
dist_y = (long)((xi - x) * inv_cos_table [view angle]);
xi_save = (int)xi;
yb_save = y_bound;
y_ray = INTERSECTION_FOUND;
casting--;
} // Конец обработки наличия блока else
{ // Прекращаем расчет по оси Y
xi += x_step[view angle];
// Вычисляем следующую возможную линию пересечения
y_bound += y_delta;
} // Конец else
}
} // Конец while
// СЕКЦИЯ 6 ////////////////////////////////
// На этом этапе мы вычислили точки пересечения с
// горизонтальными и вертикальными стенами. Теперь
// определяем, которая из них ближе
if (dist_x < dist_y)
{
// Вертикальная стена ближе
// Рассчитать масштаб и умножить его на поправочный
// коэффициент для устранения сферических искажений
scale = (int)(cos_table[ray]/dist_x);
// Отсечь фрагмент текстуры
if (scale>(MAX_SCALE-1)) scale=(MAX_SCALE-1);
scale_row = scale_table[scale-1];
if (scale>(WINDOW_HEIGHT-1))
```

```
{
sliver_clip = (scale-(WINDOW_HEIGHT-1)) >> 1;
scale=(WINDOW_HEIGHT-l) ;
}
else
sliver_clip =0;
sliver_scale = scale-1;
// Установить параметры для ассемблерной процедуры
sliver_texture = object.frames[x_hit_type];
sliver_column = (yi_save & 0x003F);
sliver_top     = WINDOW_MIDDLE - (scale >> 1);.
sliver_ray     = ray;
// Отобразить фрагмент текстуры
Render_Sliver_32();
} // Конец if else
// горизонтальная стена ближе
{
//Рассчитать масштаб и умножить его на поправочный
//коэффициент для устранения сферических искажений
scale = (int)(cos_table[ray]/dist_y);
if (scale>(MAX_SCALE-l)) scale=(MAX_SCALE-1);
// Выполняем отсечение
scale_row      = scale_table{scale-l];
if (scale>(WINDOW_HEIGHT-1)) {
sliver_clip = (scale-(WINDOW_HEIGHT-1)) >> 1;
scale=(WINDOW_HEIGHT-l);
}
else
sliver_clip = 0;
sliver_scale = scale-1;
// Устанавливаем параметры для ассемблерной процедуры
sliver_texture= object.frames[y_hit_type+l];
sliver_column = (xi_save & 0x003F);
sliver_top     = WINDOW_MIDDLE - (scale >> 1) ;
sliver_ray     = ray;
// Отображаем текстуру
Render_Sliver_32();
} // Конец else
// СЕКЦИЯ 7 ////////////////////////////////
// Переходим к следующему лучу
// Проверяем, не превысил ли угол 360 градусов
if (++view_angle>=ANGLE_360)
{
view_angle=0;
} // Конец if
} // Конец for
} // Конец функции
```

Новый отсекатель лучей сродни первому (файл RAY.C из шестой главы) и имеет то же самое строение. Оптимизация была проведена, скорее, на микроскопическом, нежели на макроскопическом уровне. Это означает, что я

оптимизировал программу строку за строкой вместо изменения базовой техники, Такой подход дал прекрасный результат и я получил увеличение скорости примерно на порядок. Я использовал множество вычислений с фиксированной запятой, целые, логические операции, несколько больше справочных таблиц для увеличения скорости вычислений до приемлемого уровня. Наконец, я прибегнул к ассемблированию, чтобы получить последние несколько процентов прироста. Хотя, вообще-то, и версия чисто на Си успешно работала на 486-й машине, но на 386-й она выполнялась исключительно медленно (по крайней мере, по словам моего друга Эшвина, а я думаю, мы можем поверить ему).

Каждая секция программы выполняет ту же самую задачу, что и в предыдущей версии. Различие заключено только в оптимизации. Вся новизна состоит в использовании целых чисел и чисел с фиксированной запятой. Это, касается того места, где было достигнуто наибольшее увеличение быстродействия. Давайте сделаем обзор всех изменений, чтобы получить представление об их суммарном эффекте. Рассмотрим секцию за секцией.

Секция 1

Здесь мы не изменяли ничего.

Секция 2

Обратите внимание, как теперь вычисляется Y-пересечение: для этого используются целочисленные и логические операции. Основная идея заключена в том, что любое число, деленное нацело по модулю N, равно тому же самому числу, логически объединенному с M-1 по принципу AND. Другими словами,

```
X % 64 == X AND 63
```

Этот трюк я использовал, чтобы сократить размер программы и увеличить скорость выполнения в этой секции просто до ненормальной.

Секция 3

В этой секции используется та же самая тактика, что и в предыдущей, только данные обрабатываются на пересечениях X, а не Y.

Секция 4

Это то место, где проверяются начала дальнейших пересечений. Я оказал этой и следуюхцей секции наибольшее внимание с точки зрения оптимизации, потому что данная часть является внутренним циклом отсекателя лучей. В любой программе в первую очередь нужно оптимизировать самый внутренний цикл, азатем уже переходить к внешним. Я внимательно рассмотрел вычисления cell_x и cell_y и оптимизировал их, использовав для деления сдвиг. Наконец, все расчеты были преобразованы к формату с фиксированной запятой за исключением вычисления расстояния, в котором по-прежнему участвуют значения типа FLOAT.

Секция 5

В этой секции применена та же самая оптимизация, что и в предыдущей. Разница только в том, что здесь рассматривается другая ось.

Секция 6

Эта секция изменена совсем чуть-чуть:

- Во-первых, кардинально изменено вычисление scale. Я провожу предварительное умножение масштаба изображения и разрешения совместно, что приводит к получению одного массива. Это экономит одну операцию умножения;
- Во-вторых, я перекомпоновал текстурные элементы и уменьшил их размер;

Далее появился фрагмент кода, проводящий отсечение по вертикали. Он отсекает каждую вертикальную линию по границам игрового окна и передает результаты расчетов вместе с исходным масштабом механизму отобра жения текстур.

Отображение текстур - это ассемблерная процедура, получающая входные данные через глобальные переменные программы. Это было сделано, для скорости. Текстурирование - чрезвычайно простая процедура. Когда обнаружено пересечение, точка вместе с соответствующим ей блоком 64x64 пикселя вычисляется и передается текстуре. Эта позиция (0-63) используется как индекс памяти текстуры для получения требуемого фрагмента.

Секция 7

Ничего интересного тут нет.

Я вполне доволен новым отсекателем лучей. Он работает отлично для такой простой

программы, особенно, учитывая, что разработана она была всего за три дня! Это наглядно показывает, чего можно достичь, имея хотя бы минимум предусмотрительности и планирования.

Изображение текстуры

Мы обсуждали эту тему на протяжении седьмой главы "Усовершенствованная битовая графика и специальные эффекты", так что я не собираюсь снова вдаваться в технические подробности. Но поскольку в Warlock вывод текстур применяется очень активно, стоит снова вернуться к этому вопросу. Текстуры для рисования вертикальных полосок извлекаются из битовой карты изображения, заранее подготовленной и находящейся где-то в памяти. Посмотрите па рисунок 19.3. На нем показаны текстуры, которые я нарисовал для Warlock.

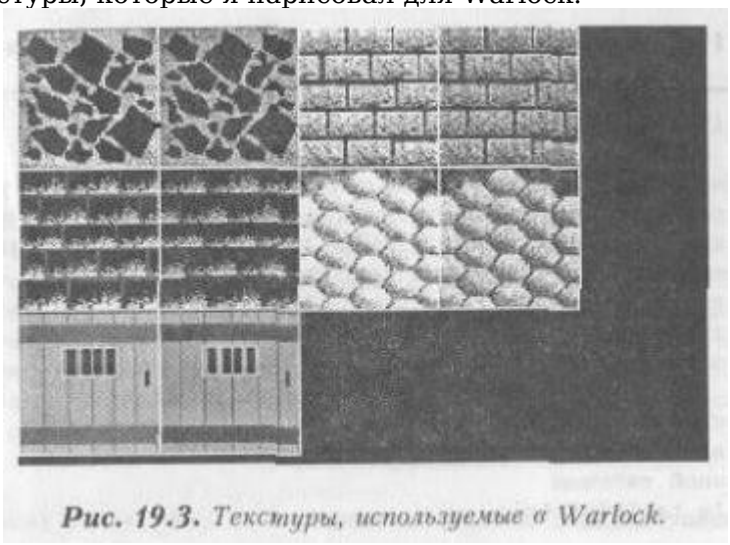


Рис. 19.3. Текстуры, используемые в Warlock.

Эти текстуры размером 64x64 пикселя были созданы с помощью редактора DPaint. Как мы узнали ранее, изображение текстуры в игре с отсечением лучей. составляется из множества вертикальных полосок. Текстура масштабируется путем преобразования исходного количества пикселей изображения к окончательному размеру. Основная задача состоит в том, чтобы выполнить эту операцию быстро. Для этого, я предложил написать программу отрисовки фрагмента на ассемблере. Это функция, рисующая отмасштабированные и текстурированные вертикальные полоски. Она работает по тому же самому алгоритму, что мы обсуждали ранее. Только сейчас я предварительно вычислил индексы масштабирования- В сущности, я заранее рассчитал все возможные масштабы для размеров стен от 1 до 220 пикселей и, подсчитав индексы масштабирования, свел их в массивную таблицу (примерно на 40K). Следовательно, если размер изображения должен составлять 100 пикселей, вы обращаетесь к сотой строке справочной таблицы. Каждая точка берется из исходной матрицы изображения, используя массив данных сотой строки и n-го столбца, где n — рисуемый пиксель. Вероятно, я объясняю так, что кажется, будто это очень трудно сделать. Жаль. Скорее всего, программа окажется более легкой для понимания! Листинг 19.2 содержит Сн-версию программы рисования фрагмента, а Листинг 19.3 содержит ее ассемблерную версию. Си-версия нуждается в добавлении механизма отсечения по границам игрового окна, но не беспокойтесь об этом. Только взгляните на внутренний цикл Си-версии и сравните его с ассемблерной версией.

Листинг 19.2. Си-версия программы для рисования фрагмента тек-стуры.

```
Render Sliver(sprite_ptr sprite,int scale, int column)
{
// Это обновленная версия функции рисования фрагмента текстуры.
// Она использует справочные таблицы с предварительно рассчитанными
// индексами масштабирования. В конце концов, я переписал ее
// на ассемблере из соображений скорости работы.
char far *work_sprite;
int far *row;
int work offset=0,offset,y,scale_off;
unsigned char data;
// Устанавливаем указатель на соответствующую строку
// справочной таблицы row = scale_table[scale];
if (scale>(WINDOW_HEIGHT-1))
{ scale_off = (scale-(WINDOW_HEIGHT-1))>>1;
scale=(WINDOW_HEIGHT-l);
sprite->y = 0;
}
// Устанавливаем указатель на спрайт для облегчения доступа
```

```
work_sprite = sprite->frames[sprite->curr_frame];
// Вычисляем смещение спрайта в видеобуфере
offset = (sprite->y << 8) + (sprite->y <<6) + sprite->x;
for(y=0;y<scale; y++)
{
double_buffer[offset] = work_sprite[work_offset+column];
offset  += SCREEN_WIDTH;
work_offset = row[y+scale_off];
} // Конец цикла for
) // Конец функции
```

Листинг 19.3. Ассемблерная версия программы рисования фрагмента текстуры (SLIVER.ASM).

```
; Эта функция является ассемблерной версией
; аналогичной функции на Си
; Она использует заранее вычисленные таблицы
; для увеличения скорости работы
;////////////////////////////
.MODEL MEDIUM, C             ; Используем модель MEDIUM
.CODE                        ; Начало сегмента кода
EXTRN double_buffer;DWORD    ; Буфер в оперативной памяти
EXTRN sliver_texture:DWORD   ; Указатель на текстуру
EXTRN sliver_column:WORD     ; Текущий столбец текстуры
EXTRN sliver_top:WORD        ; Начальная вертикальная позиция
EXTRN sliver_scale:WORD      ; Общая высота текстуры
EXTERN sliver_ray:WORD        ; Текущий столбец экрана
EXTRN sliver_clip:WORD       ; Какая часть текстуры отсекается?
EXTRN scale_row;WORD         ; Номер колонки в таблице ; масштабирования
PUBLIC Render_Sliver_32      ; Объявляем функцию общедоступной
Render_Sliver_32 PROC FAR C ; функция на Си
.386           ; использовать инструкции 80386 процессора
push si                 ; сохранить регистры push di
les di, doubie_buffer   ; установить в ES:DI адрес буфера
mov dx,sliver_column    ; загрузить номер строки в DX
lfs si, sliver_texture ; FS:SI указывает на текстуру
; offset = (sprite->y « 8) + (sprite->y « 6) + sprite->x
mov bx,sliver_top       ; умножить Y на 320
; для вычисления смещения
shl bx,8
mov ax,bx
shr bx, 2
add bx,ax
add bx,sliver_ray       ; добавить X
add di,bx
mov bx,sliver_clip        ; занести константы в регистры
mov ax,sliver_scale
add ax,bx
Sliver_Loop:                       ; основной цикл
; double_buffer [offset] = work_sprite [work_offset+column]
xchg dx,bx ; обменять содержимое BX и DX, так как
; только BX можно использовать в качестве
; индексного регистра
mov cl, BYTE PTR fs:[si+bx] ; получить пиксель текстуры
mov es:[di], cl ;поместить его на экран
xchg dx,bx       ; восстановить регистры BX и DX
mov cx,bx        ;готовимся работать с таблицей
; row = scale_table[scale]
mov dx, scale_row
shl bx,1
add bx, dx
mov dx, WORD PTR [bx] ; выбираем масштаб из массива
add dx,sliver_column
mov bx,cx
; offset      += SCREEN_WIDTH;
add di,320             ; переходим к следующей строке
inc bx                 ; инкремент счетчика
```

```
cmp bx, ax              ; работа закончена?
jne Sliver_Loop
pop di                  ; восстанавливаем регистры
pop si
ret                     ; конец работы
Render_Sliver_32 ENDP END
```

В принципе, ассемблерный вариант программы делает то же самое, что и Си-версия. Только в ассемблере используются глобальные переменные, выполняется небольшое отсечение и все работает быстрее. Я был вынужден использовать команды 386-го процессора и регистры дополнительного сегмента, чтобы ускорить работу программы, насколько возможно. Без использования регистров добавочного сегмента я был бы вынужден работать со стеком что немного уменьшило бы скорость выполнения. Теперь я должен извиниться за то, что сказал несколько раньше. Я говорил, что мы должны использовать 386 и 486 процессоры, поскольку они быстрее 8086. Это не было абсолютной правдой. Ведь применять команды, использующие 32-битовые регистры и все прочее в этом роде, можно только с привлечением ассемблера (конечно, если вы не используете DOS-расширитель вместе с 32-битным компилятором). Хотя, опять же, было бы лучше оставить эти части программы простыми и понятными.

Оттенение

Поиграв немного в Warlock, вы наверняка обратите внимание на отсутствие теней. На самом деле стены могут выглядеть слегка оттененными, но это не так. Просто я подготовил для каждой из стен по два варианта изображения:

- Для вертикальных стен я использую одну версию текстуры;
- Для горизонтальных стен я применяю просветленное изображение текстуры.

Изначально в программе было оттенение, но это уменьшило скорость ее работы примерно на 5 процентов. Но не это волновало меня больше всего. Основные проблемы, связанные с используемой техникой оттенения, возникли с палитрой. Я должен был разбить ее на части внутри зон одного и того же цвета с различными оттенками, но решил, что это было бы чересчур сложно для вас. Если вы хотите иметь оттенение, вы можете создать свою собственную палитру и использовать технические приемы, описанные в шестой главе, «Третье измерение», чтобы выбрать надлежащий оттенок в зависимости от таких параметров, как угол и расстояние.

Использование ассемблера

Использование ассемблера только иногда является последним средством увеличения быстродействия некоторой части программы до приемлемого уровня. В Warlock я написал около 100 ассемблерных строк для оптимизации только наиболее критичных по времени выполнения кусков игры. Они, конечно, относились к визуализации графики. Существует определенная норма среди программистов игр для ПК: в основном игра пишется на Си, а затем некоторые функции, практически только относящиеся к графике, переписываются па ассемблере. (Если вы вдруг обнаружили, что, применяя ассемблер для реализации искусственного интеллекта или игровой логики, с вашей программой происходит что-то неладное, стоит хорошенько подумать, не вернуться ли к Си.) Я почти уверен, что лучшие мировые программисты могли бы сделать вашу и мою программы на Си значительно быстрее и без использования ассемблера. Запомните это, и когда найдете ассемблер пригодным не только для функций, связанных с графикой и низкоуровневым программированием, признайте, что надо сделать шаг назад (и шаг вперед... словно мы танцуем ча-ча-ча!) и начать сначала. Так или иначе, на ассемблере я переписал только две функции, которые перемещают содержимое дублирующего буфера в видеопамять и рисуют небо и землю (Листинг 19.5). Если уж на то пошло, их ассемблерный вариант занимает всего 5-10 строк.

Листинг 19.4. Функция, переносящая содержимое дублирующего бу-фера в видеопамять (DRAWG.ASM).

```
; Функция просто копирует дублирующий буфер в видеопамять
;Она использует 32-битовые операции и регистры
;для увеличения быстродействия
;/////////////////////
.MODEL MEDIUM,C               ; используем medium модель
; и соглашения языка Си
.CODE                         ;начало кодового сегмента
EXTRN double_buffer:DWORD     ; указатель на дублирующий буфер
PUBLIC Draw_Ground_32         ; делаем функцию общедоступной
Draw_Ground_32 PROC FAR C     ; функция поддерживает-соглашения
; по вызову для языка Си и является ; дальней
.386            ; использовать инструкции процессора 80386
push di                       ; сохраняем регистр DI
```

```
cld                             ; сбрасываем флаг направления
les di, double_buffer           ; загружаем в ES:DI адрес буфера
xor eax,eax                     ; обнуляем регистр EAX
mov cx,320*76/4                 ; заполняем 76 строк
rep stosd                       ; делаем это
mov eax,01E1E1E1Eh              ; загружаем в EAX код серого цвета
mov cx,320*76/4                 ; нам надо заполнить 76 строк
rep stosd                       ; делаем это
pop di                          ; восстанавливаем DI
ret                            ; конец
Draw_Ground_32 ENDP END
```

Листинг 19.5. функция визуализации неба и земли (RENDERB.ASM).

```
; Эта функция рисует землю и небо. В ней используются
;  32-битовые регистры и инструкции.
;//////////////////////////////////
.MODEL MEDIUM,C                 ; использовать medium модель и соглашение Си
.CODE                           ; начало кодового сегмента
EXTRN double_buffer:DWORD       ; адрес дублирующего буфера
EXTRN video_buffer:DWORD        ; адрес видеобуфера
PUBLIC Render_Buffer_32         ; делаем функцию общедоступной
Render_Buffer_32 PROC FAR C ; функция поддерживает соглашения
; Си и является дальней
.386                    ; использовать инструкции 80386 процессора
push ds                         ; сохранить регистр сегмента данных
cld                           ; сбрасываем флаг направления
lds si, double_buffer           ; в регистры источника (DS:SI)
; загружаем адрес буфера
les di, video_buffer            ; в регистры приемника (ES:DI)
;загружаем адрес видеопамяти
mov cx,320*152/4                ; определяем количество двойных слов
rep movsd                       ; копируем
pop ds                          ; восстановить регистр сегмента данных
ret                             ; сделано!
Render_Buffer_32 ENDP
END
```

Цикл игры

Как я сказал, Warlock едва ли является полноценной игрой. Фактически, в ней вы можете только погулять, обозревая окрестности. Я создал механизм работы с трехмерной графикой, а вы сделайте из него игру. Но несмотря ни на что, в моей разработке все же присутствует некоторый набросок сюжета, который мы и можем обсудить.

Вся программа игры сосредоточена в исходном модуле WARLOCK.C. Я собрал все в одном модуле вместо того, чтобы разбивать программу на части, чтобы вам легче было ее понять и добавить то, что вы сочтете нужным.

Игра основывается на структуре управляемых событий. Под управляемыми событиями я подразумеваю, что программа реагирует на действия игрока, использующего только ввод с клавиатуры. Когда игрок нажимает клавишу, изменяются местоположение персонажа, угол зрения, или то и другое вместе. Полученная информация обрабатывается и отображается в следующем кадре. Игра имеет следующую структуру событий:

1. Инициализация; изображение и звуки загружены, справочные таблицы созданы- Также установлен новый обработчик прерывания клавиатуры, позволяющий обрабатывать одновременное нажатие нескольких клавиш.

2. Опрос клавиатуры. Если какая-токлавиша была нажата или отпущена, это отражается в справочной таблице изменения статуса клавиатуры. Полученная информация используется для выбора направления движения.
3. Если на предыдущем шаге определено нажатие клавиш перемещения, то вычисляются новые координаты персонажа. Если они допустимы, то все в порядке, в противном случае персонаж возвращается на первоначальную позицию.
4. Проверяется нажатие клавиши Пробел (с ее помощью можно открывать двери). Если Пробел нажат и дверь находится точно напротив игрока, выполняется функция, отвечающая за открывание двери. Она последовательно выводит кадры анимации распахивающейся двери и открывает в этом месте проход.
5. В Warlock используются звуки, имитирующие вой ветра, стоны, рычание. Если соответствующая переменная принимает определенное значение, случайным образом выбирается и проигрывается один из этих звуков.
6. Ожидание вертикальной синхронизации экрана (описанной в седьмой главе). Это придает ходу игры определенный ритм и позволяет свести к минимуму мерцание изображения при его обновлении.
7. Построение изображения отсечением лучей.
8. Возврат к шагу 2.

Игра будет пуста без сюжета и противников. Это те вещи, которые вам необходимо ввести в нее. Я допускаю, что в моей версии игры маловато спрайтов, Но если бы я увеличил их количество, программу пришлось бы еще больше оптимизировать, переписывая отдельные части на ассемблере. Мне же хотелось оставить ее простой и позволить вам доработать ее самостоятельно. У вас уже есть все необходимое, чтобы добавить спрайты в игровое пространство и я дам вам несколько советов (в разделе «размещение объектов в пространстве»), чтобы нацелить вас в нужном направлении. А пока поговорим о формировании игрового пространства.

Игровое пространство

Игровое пространство или игровое поле представлено двухмерной матрицей размером 64x64 элемента. Она содержит целые числа, которые представляют собой различные виды текстуры, отображаемые на стенах элемента. Warlock получает эти данные из файла RAYMAP.DAT.

Для создания игрового пространства Warlock можно воспользоваться программой WarEdit (которую мы создали в пятнадцатой главе, «Инструментальные средства»), но прежде потребуется ее слегка модифицировать. Опять же, я хочу, чтобы это сделали вы сами. WarEdit создает поле размером 200x200 элементов, а для Warlock требуется создать массив 64x64. Поэтому вам потребуется изменить размерность и целочисленные коды объектов,

На рисунке 19.4 показана матрица игрового пространства, в котором вы движетесь.

Как видите, игровое пространство составляется из чисел, имеющих следующее толкование:

1 - Использовать текстуру № 1 для этих блоков;
3 - Использовать текстуру № 2 для этих блоков;
5 - Использовать текстуру № 3 для этих блоков;
7 - Использовать текстуру № 4 для этих блоков.

Номера текстур в действительности являются индексами, обозначающими последовательность загрузки в память. На самом деле каждая текстура имеет два оттенка, то есть существуют текстурные пары: (1,2), (3,4), (5,6) и т. д. Изображения текстур находятся в файле WALLTEXT.PCX. Они последовательно считываются слева направо и сверху вниз.

В сущности, это все о представлении игрового пространства. Не так уж и много. В WarEdit'e, если вы помните, мы использовали значительно больше определений для описания различных элементов и их сущности. В Warlock мы видим, что многое из этой начинки не является в настоящий момент необходимым. Однако если вы модернизируете ядро игры, эта начинка вам пригодится.

Одна из интересных вещей насчет Warlock заключена в демонстрационном режиме. Обсудим его механику.

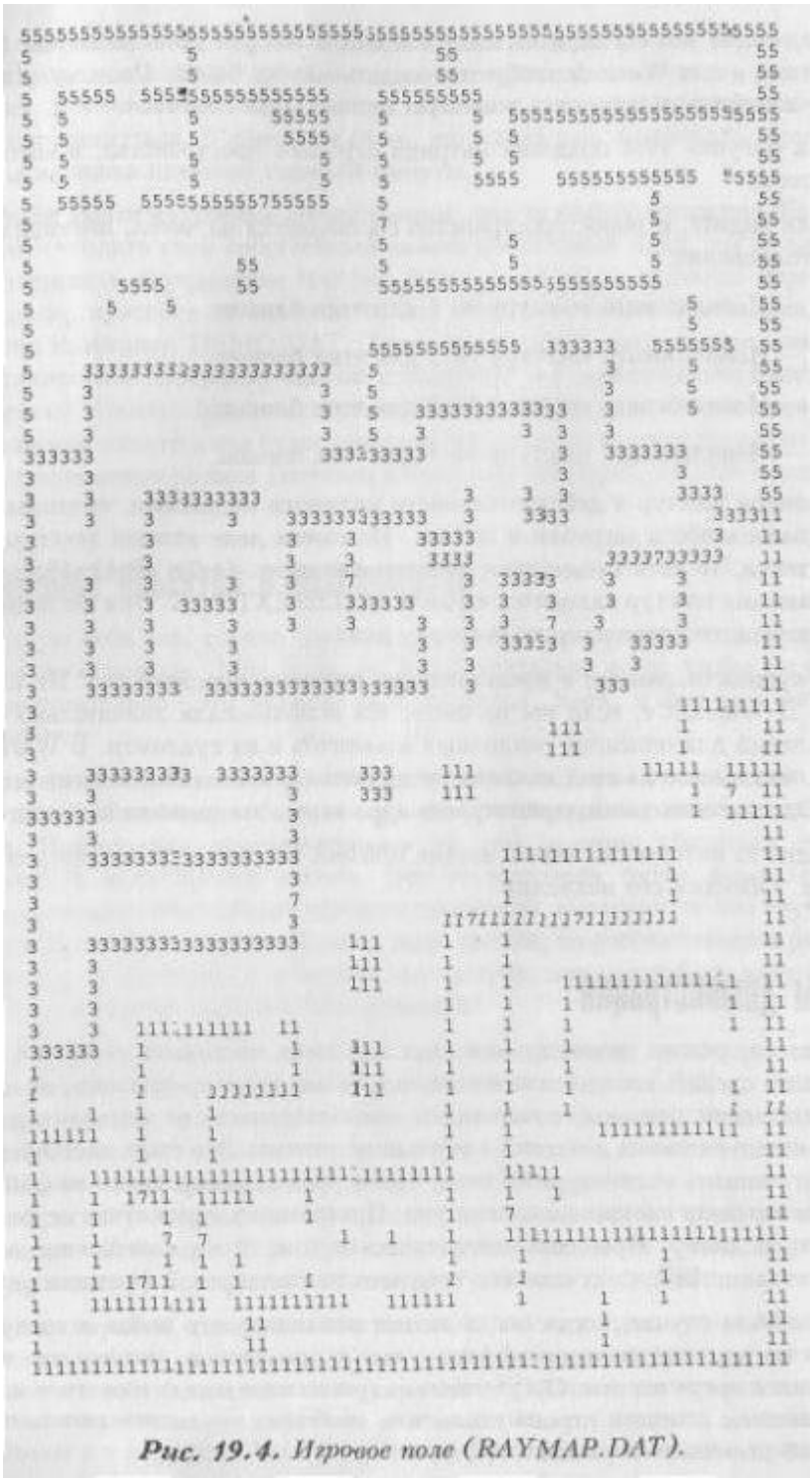


Рис. 19.4. *Игровое поле (RAYMAP.DAT).*

Режим демонстрации

Признаться, режим демонстрации стал для меня настоящей головной болью. Я пытался сделать его максимально простым для конструирования, но никакой из технических приемов, которыми я воспользовался, не позволил добиться воспроизведения моих действий в реальном времени. Это была настоящая беда! Я хотел записать клавиатурный ввод, чтобы позже проиграть его из файла, как если бы это была настоящая клавиатура. Программа в этом случае не смогла бы заметить подмену. Проблема заключалась в том, что я совершенно забыл о существовании ISR, получающего ввод вместо стандартной функции getch ().

В любом случае, когда после записи клавиатурного ввода я воспроизвел его, то не получил того же эффекта, что и при записи, потому что ввод не управлялся прерыванием. Отсутствие синхронизации между нажатием клавиши и изменением позиции игрока позволило свободно проходить сквозь стены и дало еще несколько странных эффектов.

Чтобы справиться с этой проблемой, для синхронизации игры я выбрал момент начала действий игрока, а не момент прихода команды на выполнение действия. Я отмечал тот момент, когда персонаж двинулся, а не когда ему было приказано двинуться. В конечном счете, это нормально заработало. Демонстрационная заставка крутится порядка минуты.

Чтобы выйти из режима демонстрации, просто нажмите клавишу Esc. Если вы хотите создать спой собственный демонстрационный клип, вы можете раскомментировать определение MAKING_DEMO и изменить значение переменной demo_mode, присвоив ей значение 0. Это позволит создать демонстрационный клип под названием DEMO.DAT. Затем вы должны отменить все изменения, откомпилировать программу заново и запустить ее: демонстрация будет вашей собственной. Однако будьте осторожны! В программе отсутствует проверка на переполнение памяти и она будет серьезно повреждена, если вы превысите длину демонстрационного ролика (которая рассчитана примерно на 1000 команд).

Размещение объектов в пространстве

Я чувствую себя так, словно прощаюсь с вами, так и не разместив спрайты в игровом пространстве. Мне жаль, но я действительно хочу, чтобы вы сделали это самостоятельно. Это укрепит ваш характер. Хотя, я дам вам некоторые разъяснения.

Рисование спрайтов в игровом пространстве отделено от механизма трассировки лучей. Это самые обычные спрайты, и их можно размещать где и как угодно. Прежде чем визуализировать их, мы должны убедиться, что они находятся в поле зрения игрока. Это выполняется путем нахождения их местоположения относительно игрока и проверкой, попадают ли они внутрь угла зрения игрока. Если они попадают в ноле зрения, то рассчитывается ракурс, в котором игрок их видит. Как только это сделано, они могут быть визуализированы. Но как насчет скрытых поверхностей?

Во время трассировки лучей вы храните расстояние до каждого из фрагментов. Однако массив таких данных — это на самом деле Z-буфер. Эти данные используются для сравнения расстояний до каждой из вертикальных линии спрайта. Вычисляется расстояние от спрайта до игрока. Затем во время рисования каждого из вертикальных фрагментов спрайта, расстояние между игроком и этим фрагментом сравнивается с рассчитанными во время трассировки лучей значениями в Z-буфере. Если фрагмент спрайта находится ближе, он рисуется. В противном случае, он перекрыт стеной и рисовать его не надо.

Итак, спрайты ничем не отличаются от прочих трехмерных объектов за исключением того, что они выводятся всегда перпендикулярно лучу зрения (что упрощает их визуализацию и тексту текстуирование). Однако все механизмы работы с трехмерными многоугольниками, которые рассматривались в шестой главе применимы и к спрайтам. Они выглядят размещенными в игровом пространстве после того, как выполнена отриеовка всех остальных элементов, однако в действительности, отриеовка спрайтов и остальных элементов совершенно независимы друг от друга. .

Быстрее, быстрее, еще быстрее

Перечислю некоторые вещи, которые можно сделать для ускорения трассировки лучей.

- Во-первых, начальная часть (секции 2 и 3) могут быть ускорены примерно на один процент за счет проведения некоторых предварительных вычислений;
- Во-вторых, можно убрать последнюю оставшуюся операцию с плавающей точкой;
- В-третьих, можно разделить трассировку по осям X и Y по разным секциям, что сэкономит несколько операций сравнения;
- В-четвертых, можно чередовать трассировку лучей и отрисовку экрана, трассируя лучи во время ожидания регенерации экрана;
- Наконец, вместо режима 13h можно использовать так называемый режим X, имеющий

разрешение 320x240 и являющийся наиболее быстрым из всех.
Но, это уже совсем другая история...

Несколько слов напоследок

Все, я больше не хочу говорить об играх для ПК. Теперь у вас есть необходимая техническая информация. Все остальное зависит от вашего воображения. Надеюсь, вы сможете написать совершенно невероятную игру, которая доставит другим удовольствие. Лично я делаю игры именно по этой причине. Но какие бы мотивы ни руководили вами, я надеюсь, что в вашем лице увижу новых исследователей виртуальных миров. И если мы встретимся где-нибудь в Туманности Андромеды, вам лучше держать ухо востро!

А может быть ваши путешествия будут магическими...

Любые вопросы, замечания и т. д. присылайте по адресу:

Andromeda Industries
P.O. Box 641744
San Jose, CA 95164-1744